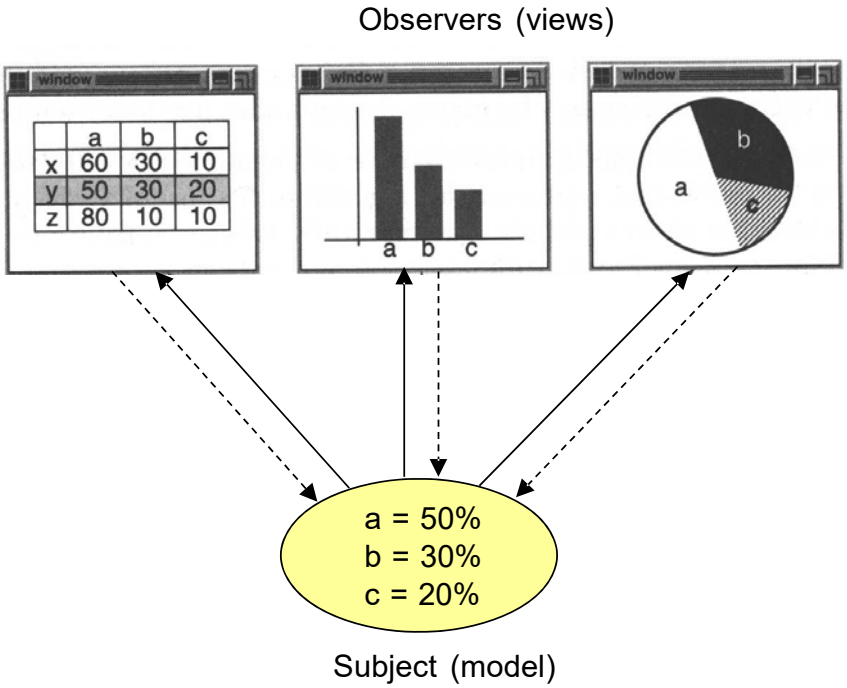
Object-Oriented Analysis and Design using UML and Patterns

GRAP Patterns (I)

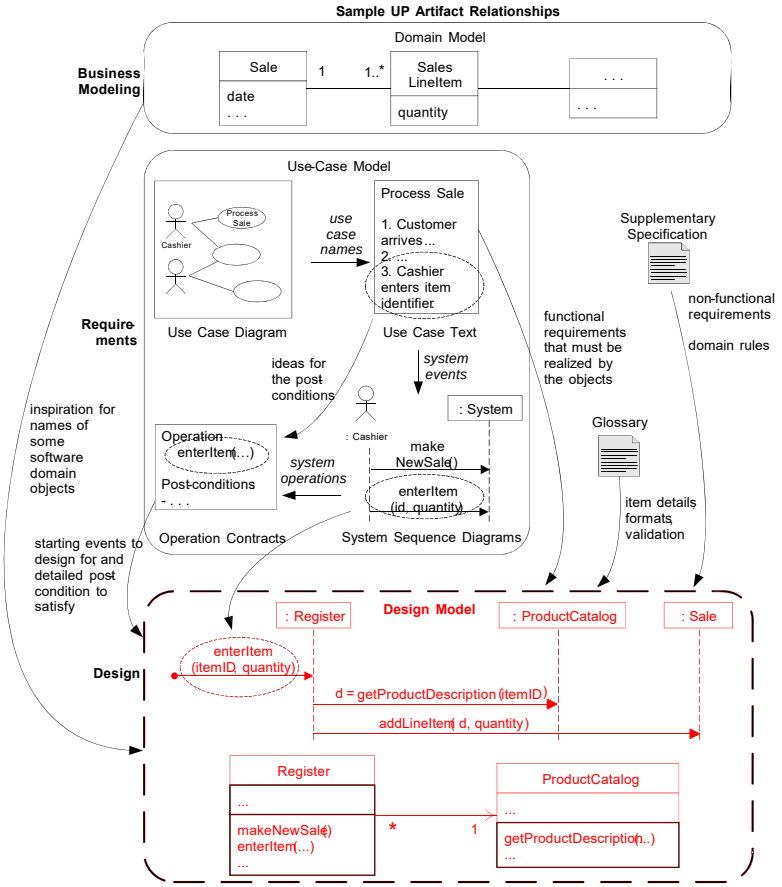


From Analysis to Design

Objectives

Understand dynamic and static object design modeling

Where are we?



Analysis Phase Conclusion

- The analysis phase emphasizes an understanding of the requirements, concepts, and system operations.
- Investigation and analysis focuses on questions of *what* -- what are the processes, concepts and so on.

Analysis Phase Artifacts

ANALYSIS ARTIFACTS	QUESTIONS ANSWERED
Use cases	What are the domain processes? What are functional requirements?
Domain Model	What are the concepts, terms?
System sequence diagrams	What are the system events and operations?
Contracts	What do the system operations do?

Design Phase

- The design phase focuses on questions of *how* -- how the system fulfills the requirements.
- The design phase emphasizes a *logical solution* from the perspective of objects (and interfaces).
- The heart of the solution is the creation of the following two diagrams, which are part of the Design Model in UP:
  - **Interaction diagrams**
    - how objects communicate, dynamic aspects
  - **Class diagrams**
    - static object relationships, static aspect

Responsibilities

Obligations of an object in terms of its behavior.

- "Doing" responsibilities:
  - doing something itself
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- "Knowing" responsibilities:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

Responsibilities and Methods

- Responsibilities are assigned during OOD.
- Responsibilities are implemented using methods which either act alone or collaborate with other methods and objects.
  - Draw itself on the canvas.
  - Print a sale.
  - Provide access to relational database.

Increasing levels of granularity

Responsibility Assignment & Interaction Diagrams

- Interaction diagram shows the design decisions in assigning responsibilities to object(s).
  - The design decisions are reflected in what messages are sent to which classes of objects.
- The amount of time and effort spent on their generation, and the careful consideration of responsibility assignment, should absorb a significant percentage of the design phase of a project.
- **Creation of interaction diagram requires knowledge of responsibility assignment principles and codified patterns to improve the quality of their design.**

GRASP Patterns (1)

Objectives

Learn to apply five of the GRASP principles or patterns

GRASP Patterns

General Responsibility Assignment Software Patterns

- Information Expert (or Expert)
- Creator
- High Cohesion
- Low Coupling
- Controller

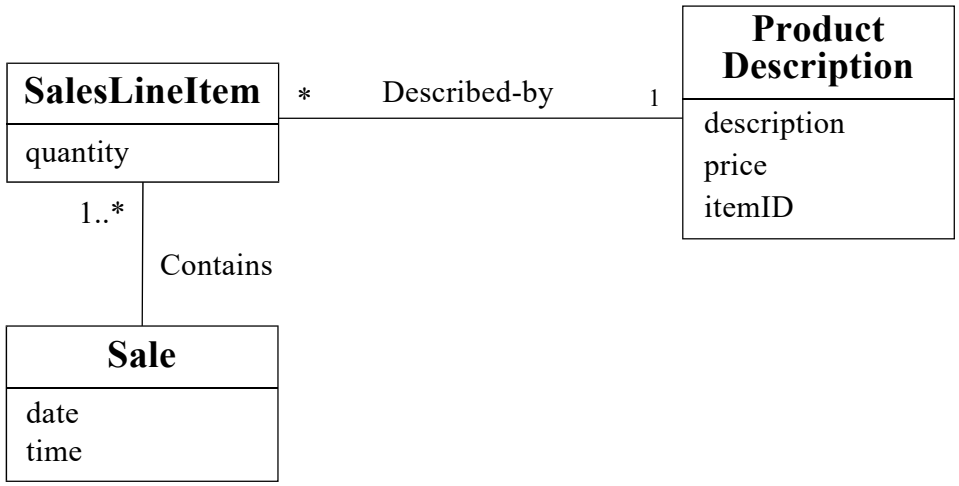
Information Expert Pattern

- Problem
  - What is the most basic principle by which responsibilities are assigned in OOD?
- Solution
  - **Assign a responsibility to the information expert -- the class that has the *information* necessary to fulfill the responsibility.**

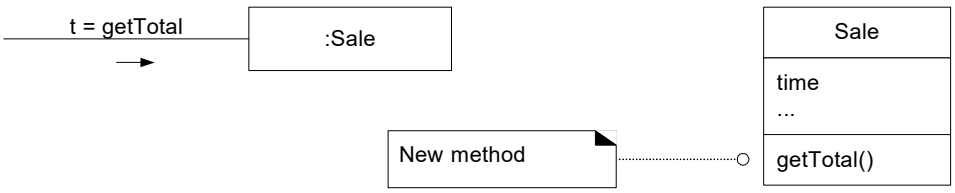
Example: Expert Pattern

- We need to know the grand total of the sale. Then

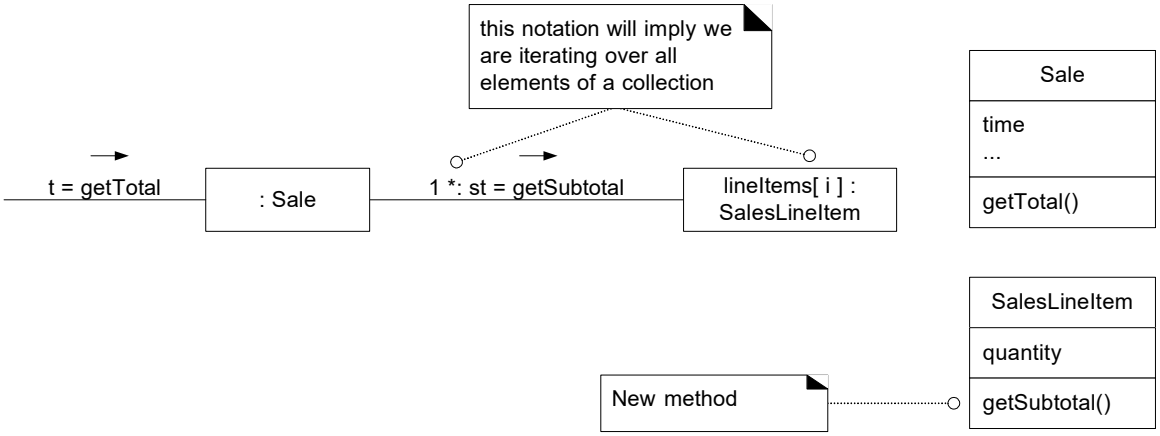
  Who should be responsible for knowing the grand total of the sale?



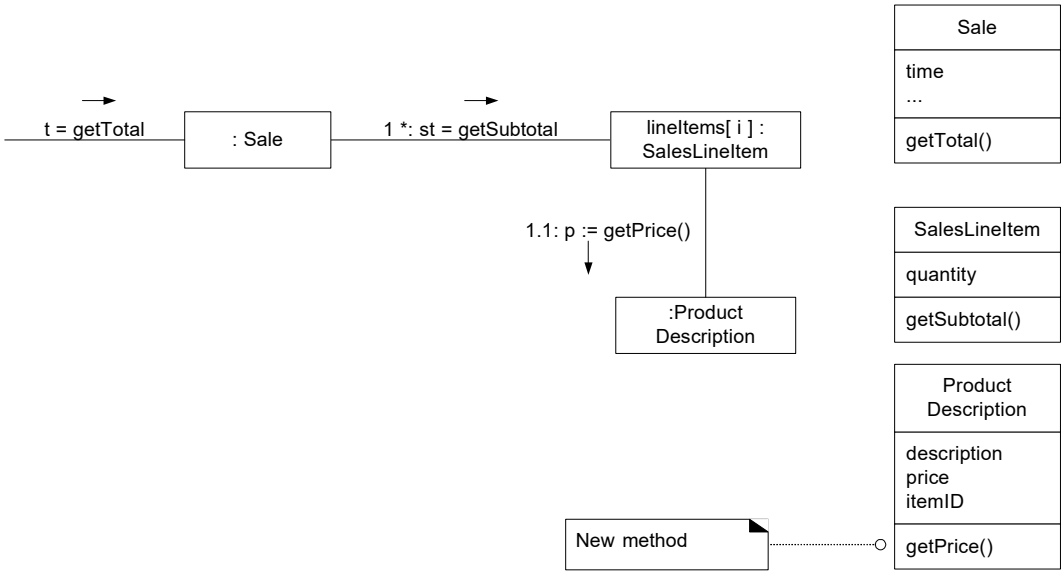
Example: Expert Pattern
(Cont'd)



Example: Expert Pattern
(Cont'd)



Example: Expert Pattern
(Cont'd)

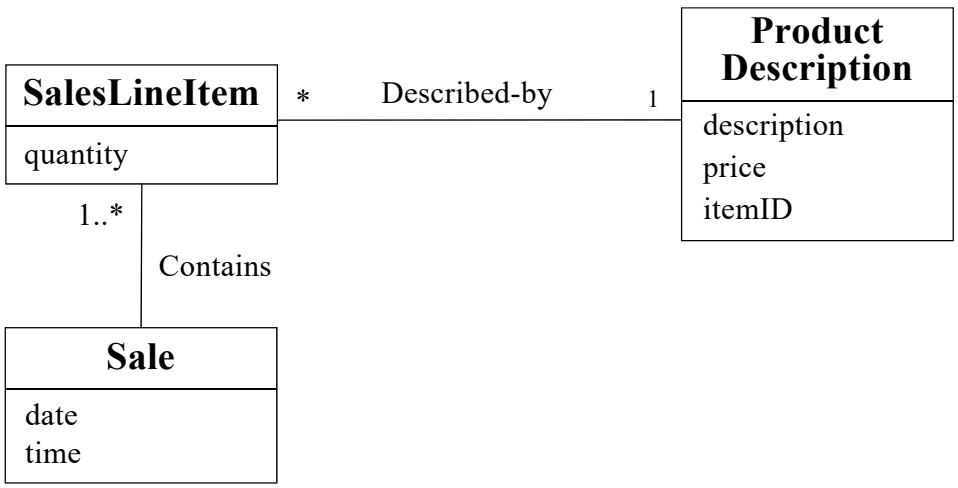


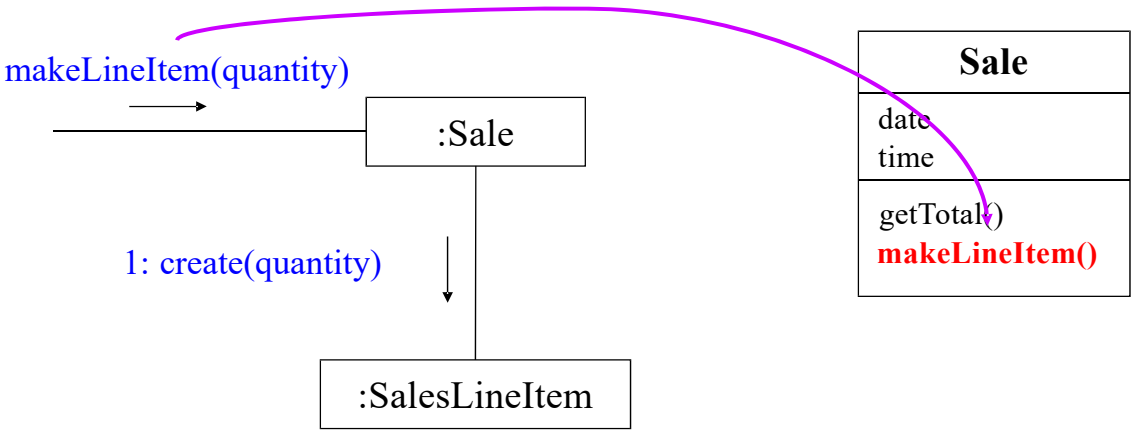
Creator Pattern

- Problem
  - Who should be responsible for creating a new instance of some class?
- Solution
  - **Assign class B the responsibility to create an instance of class A if one of the following is true:**
    - **B *aggregates* A objects.**
    - **B *contains* A objects.**
    - **B *records* instances of A objects.**
    - **B *closely uses* A objects.**
    - **B *has the initializing data* that will be passed to A when it is created.**

Example: Creator Pattern

- *Who should be responsible for creating the SalesLineItem instance?*





Low Coupling Pattern

- Problem
  - How to support low dependency and increased reuse?
- Solution
  - **Assign a responsibility so that coupling remains low.**

Low vs. High Coupling

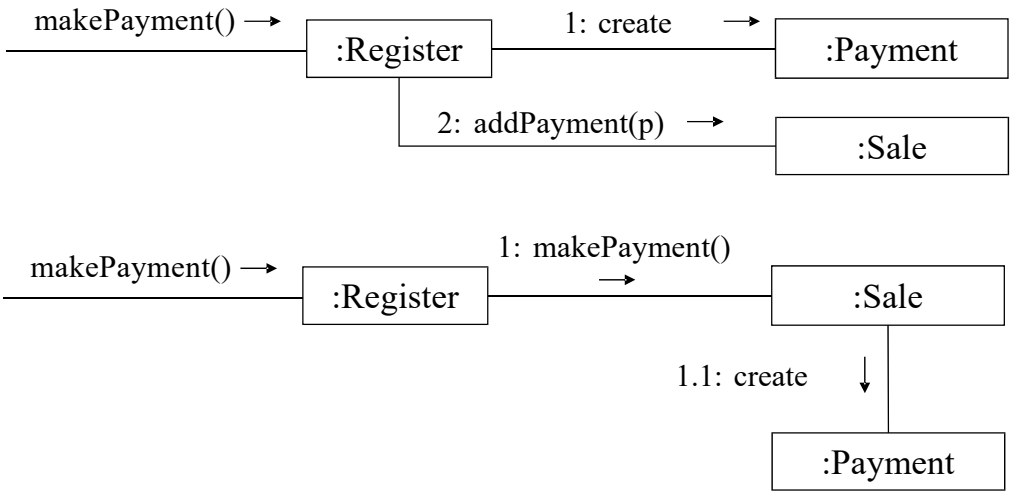
- Coupling represents the degree of dependencies of a class on other classes.
- Low coupling supports the design of classes that are
  - more independent, more reusable, and easier to understand in isolation ==> higher productivity
  - less effected by changes in other components.

Common Forms of Coupling

- *TypeX* has *TypeY* as its instance variable.
- *TypeX* has a method which has a parameter of type *TypeY*, or a return value of type *TypeY*, or a local variable of type *TypeY*.
- *TypeX* is a direct or indirect subclass of *TypeY*.
- *TypeY* is an interface, and *TypeX* implements that interface.

Example: Low Coupling Pattern

- We need to create a *Payment* and associate it with the *Sale*. Who should be responsible for this?



High Cohesion Pattern

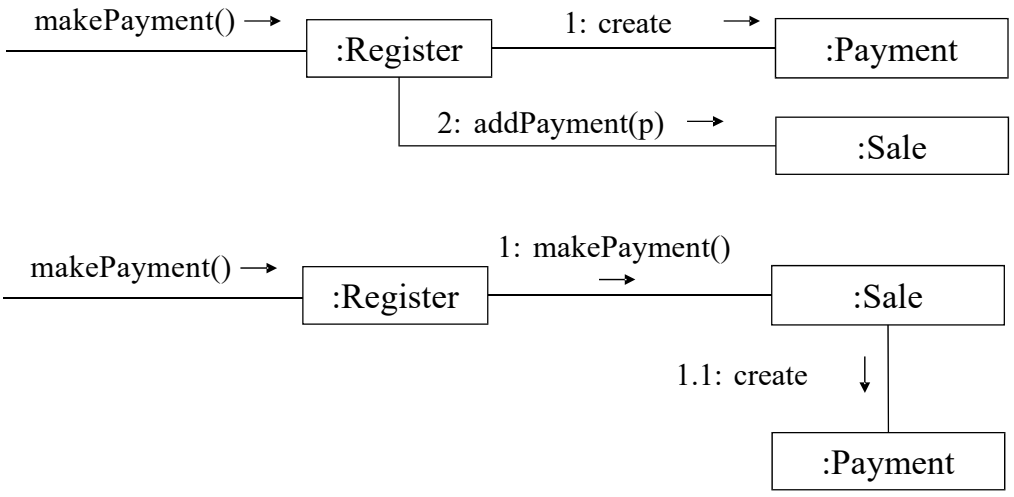
- Problem
  - How to keep complexity manageable?
- Solution
  - **Assign a responsibility so that cohesion remains high.**

Low vs. High Cohesion

- (Functional) cohesion is a measure of how strongly related and focused the responsibilities of a class are.
- A class with highly related responsibilities, and which does not do a lot of work, has high cohesion.
- A class with low cohesion has the following problems:
  - hard to comprehend
  - hard to reuse
  - hard to maintain
  - delicate; constantly effected by change

Example: High Cohesion Pattern

- We need to create a *Payment* and associate it with the *Sale*. Who should be responsible for this?



High Cohesion & Low Coupling

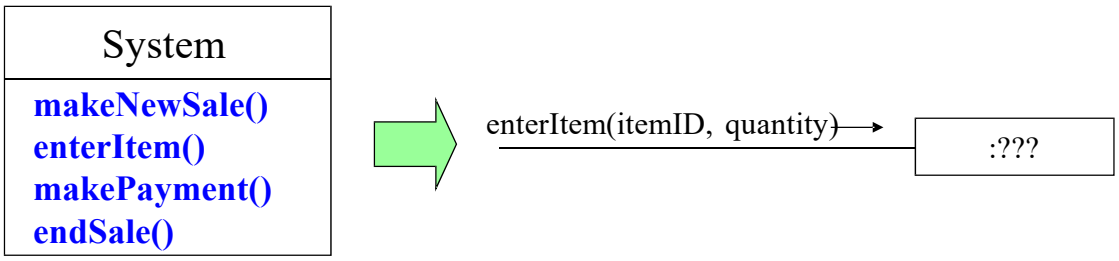
- The *high cohesion and low coupling* principles must be kept in mind during all design decisions.
- They are *evaluative patterns* which a designer applies while evaluating all design decisions.

The Controller Pattern

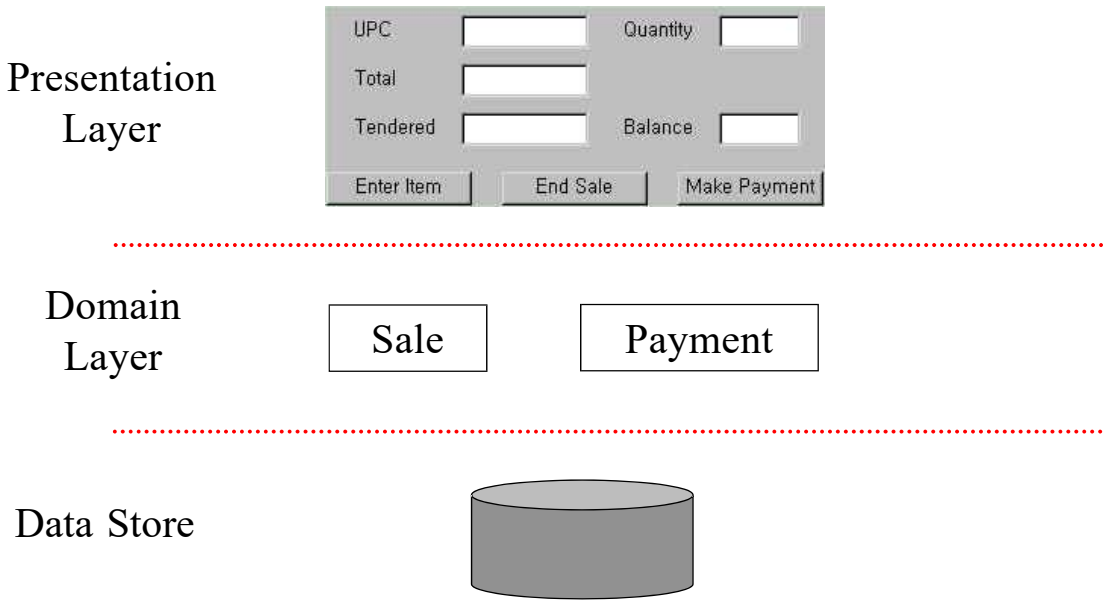
- Problem
  - Who should be responsible for handling a system event?
- Solution
  - **Assign the responsibility for handling a system event to a class representing one of the following choices:**
    - **Represents the overall system, device, or subsystem (façade controller)**
    - **Represents an artificial handler of all system events of a use case, usually named "<UseCaseName>Handler", or "<UseCaseName>Coordinator", "<UseCaseName>Session", or (use-case controller or session controller)**

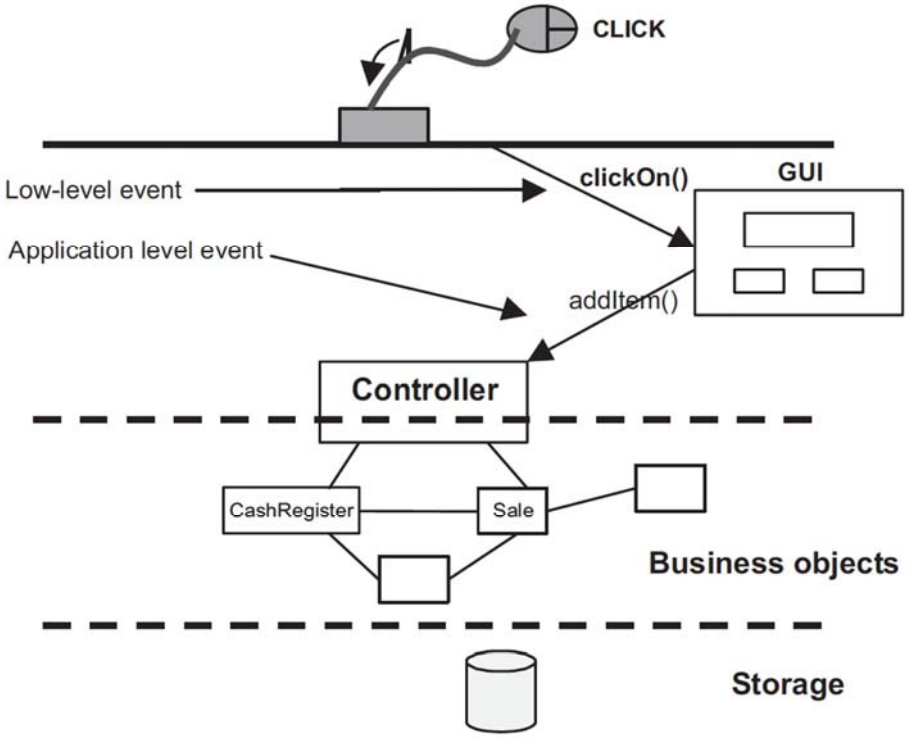
From System to Controllers

- During system behavior analysis, system operations are assigned to the type System, to indicate they are system operations. However, this does not mean that a class named System fulfills them during design.
- Rather, during design a controller class is assigned the responsibility for system operations.



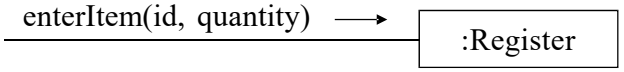
Typical Layered Architecture

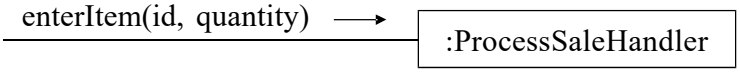




Controllers

- A controller is a non-GUI object responsible for handling a system event. A controller defines the method for the system operations.
- In case of use-case handler, use the same controller for all the system events in the same use case. ***Why?***





Which one to choose?

Bloated Controllers

- A bloated controller has a low cohesion -- unfocused and handling too many responsibilities.
- Signs of bloating include:
  - There is only a single controller receiving all system events in the system, and there are many of them: role or façade controller.
  - The controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work.
  - A controller has many attributes, and maintains significant information about system or domain, or duplicates information found elsewhere.

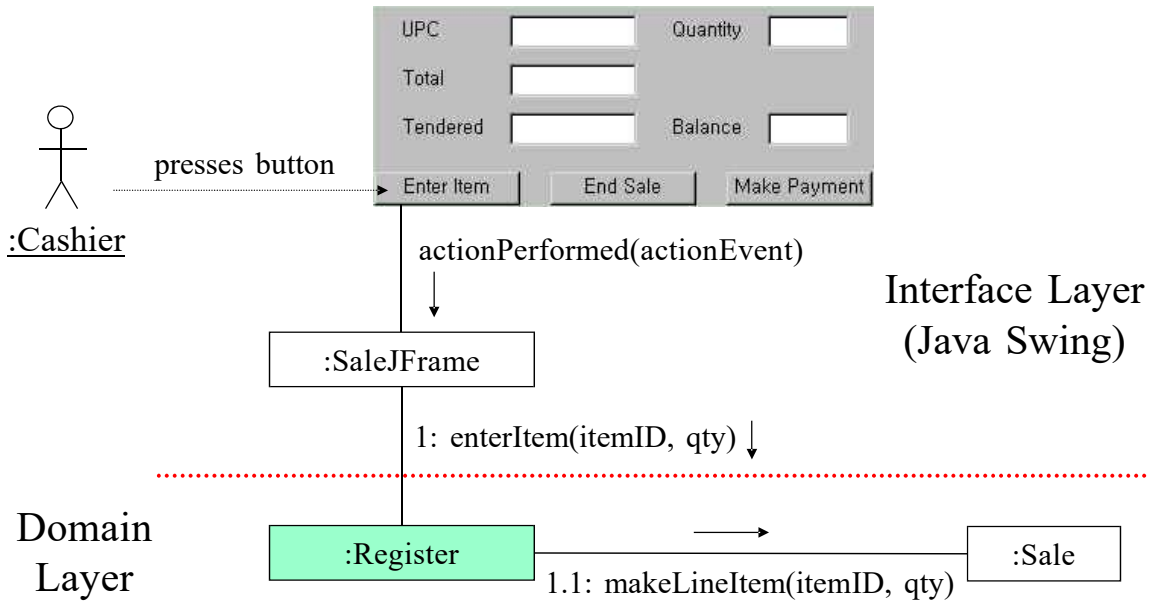
Cures to Bloated Controller

- Add more controllers.
  - In addition to façade controller, use use-case controllers.
- Design the controller so that it primarily *delegates* the fulfillment of each system operation responsibility to other objects; it coordinates or controls the activity. It does not do much work itself.

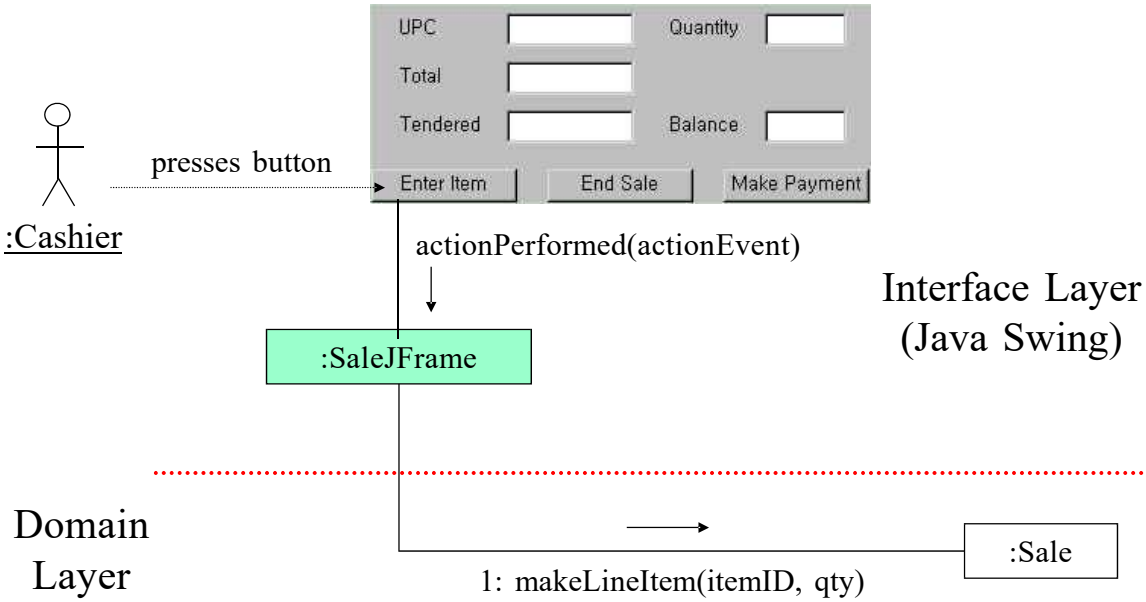
Corollary of Controller Pattern

- The GUI objects (e.g., window objects, applets, etc) in the presentation layer should *not* have responsibility for handling system events.
- It is not a good design practice to make the presentation layer have parts of domain logic which reflects business or domain process. ***Why?***

The Good

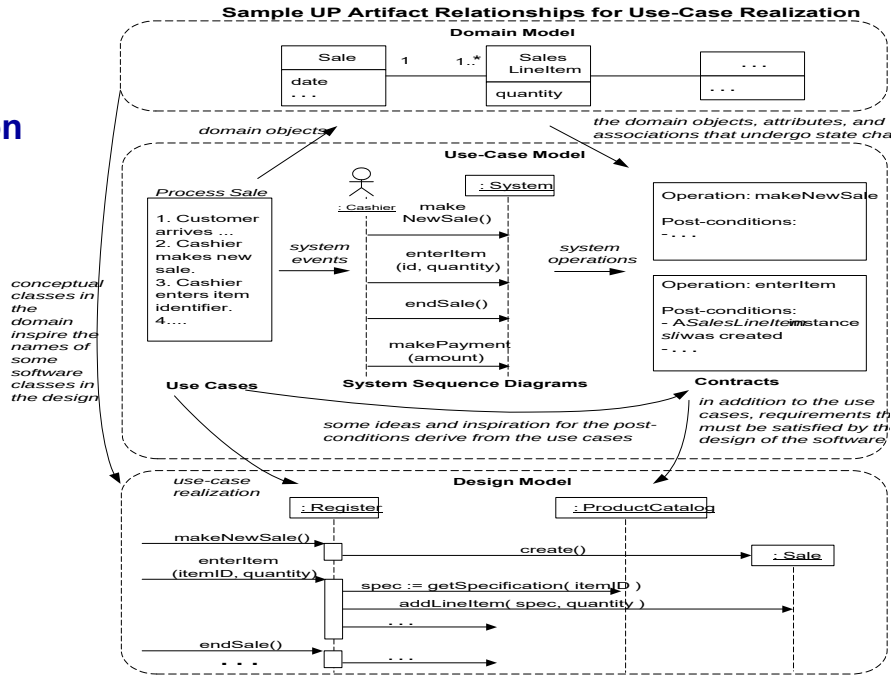


The Bad and/or The Ugly



Object-Oriented Analysis and Design using UML and Patterns

Use Case Realization



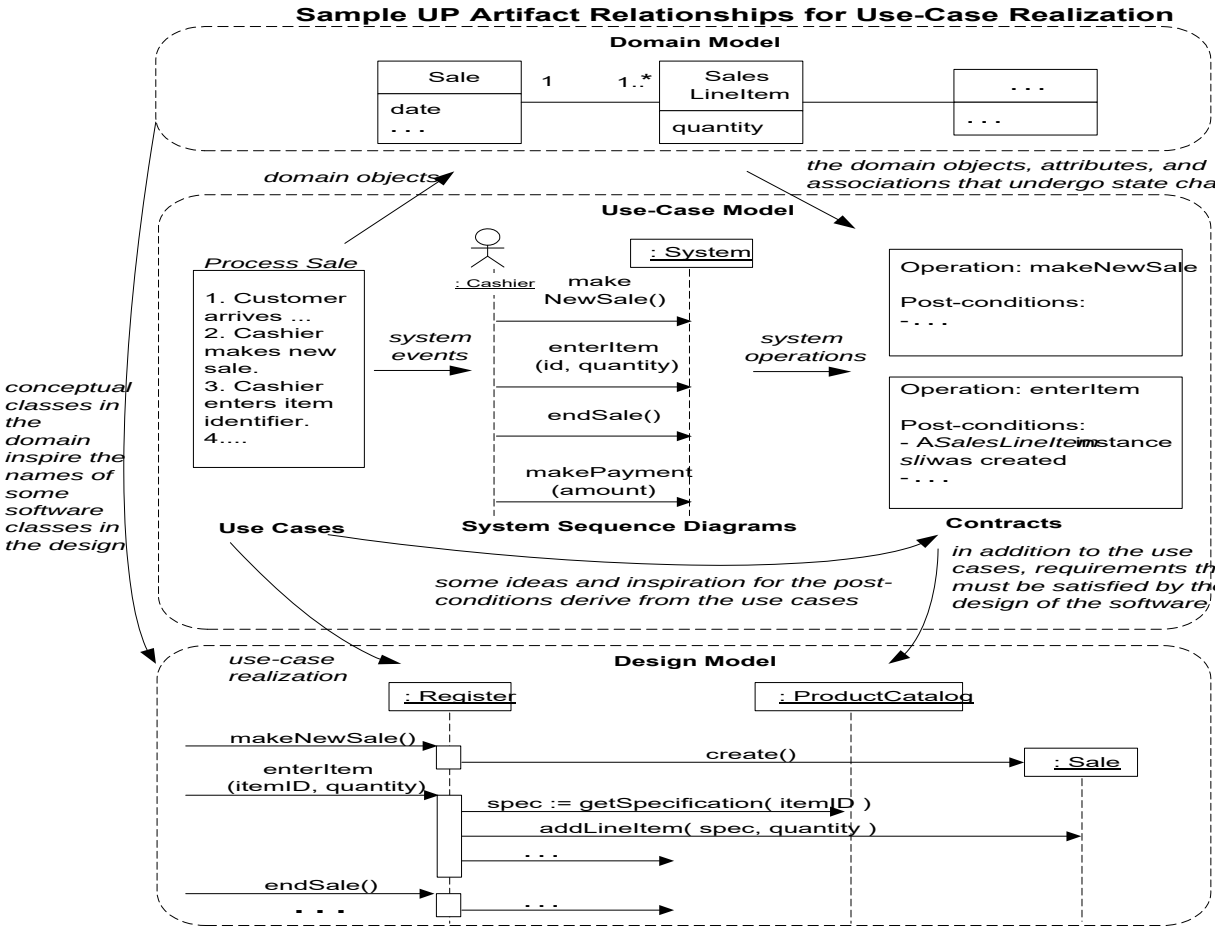
Use-Case Realization With GRASP Patterns

Objectives

Design use case realizations.

Apply GRASP to assign responsibilities to classes.

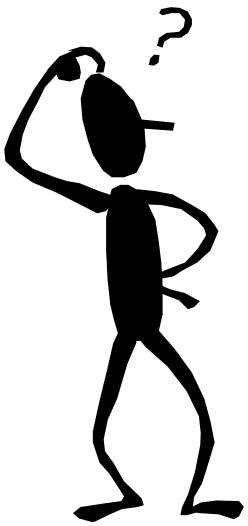
Apply UML to illustrate and think through the design of objects.

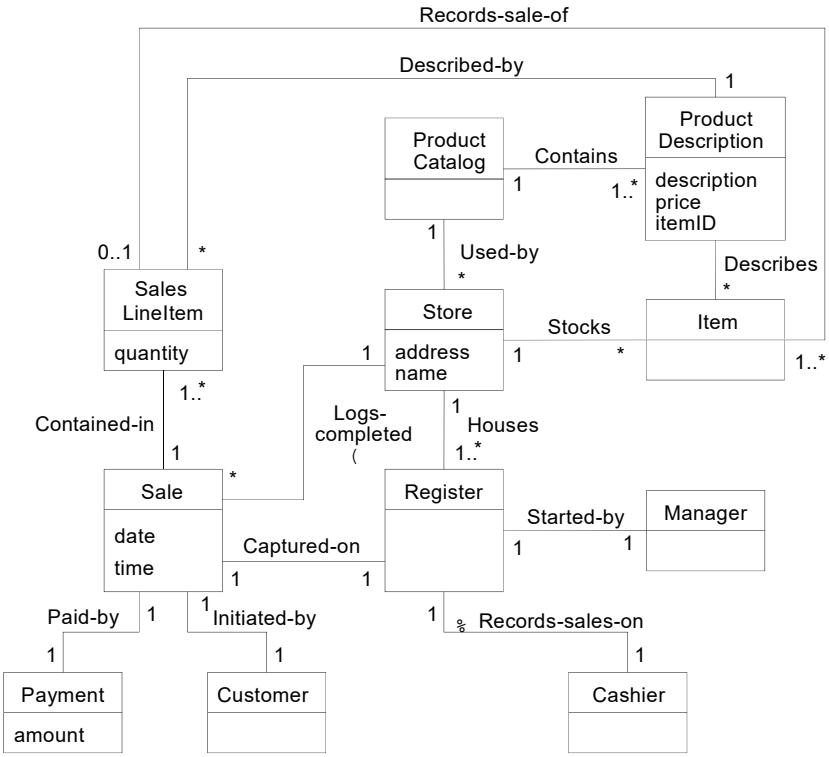


Case Study

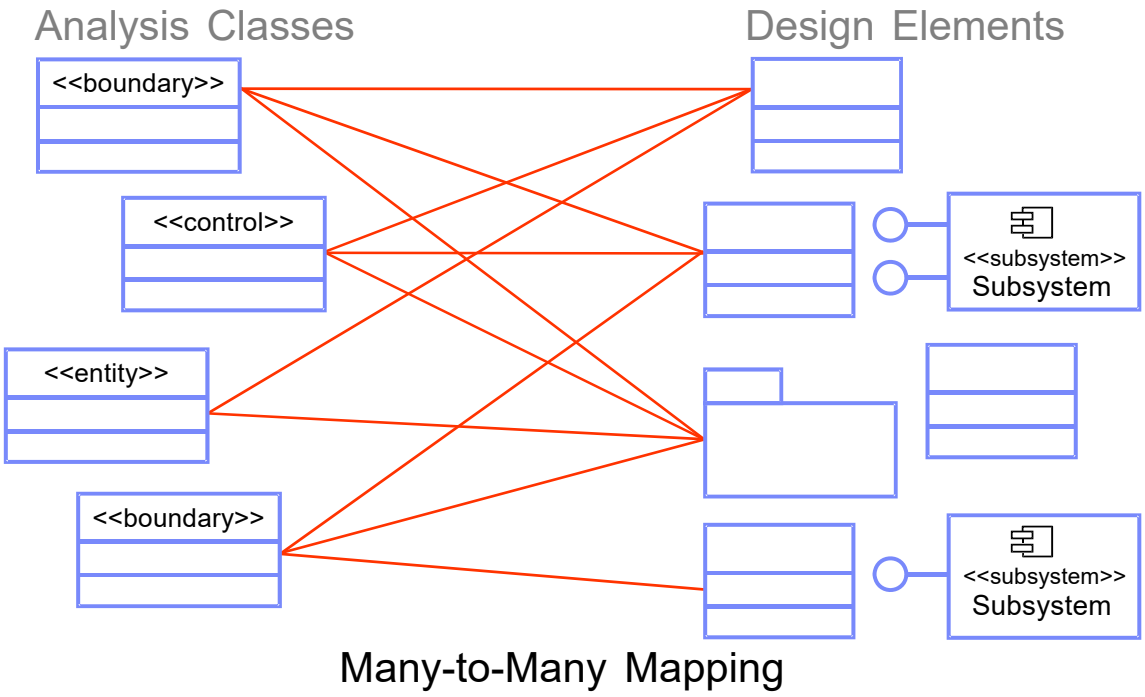
- We will consider the "*Process Sale*" and "*StartUp*"use cases and their associated system events.
  - Process Sales: makeNewSale, enterItem, endSale, makePayment
  - StartUp: startUp
- Using Controller pattern, the Register class will be chosen as the controller for handling the events.

Must the set of interacting objects be limited to this model?





From Analysis Classes to Design Elements



Analysis Classes vs. Design Elements

- Analysis classes:
  - Handle primarily functional requirements
  - Model objects from the “problem” domain
- Design elements:
  - Must also handle nonfunctional requirements
  - Model objects from the “solution” domain

Drivers When Identifying Design Elements

- Non-functional requirements, for instance consider:
  - Application to be distributed across multiple servers
  - Real-time system vs. e-Commerce application
  - Application must support different persistent storage implementations
- Architectural choices
  - For instance, .NET vs. Java Platform
- Technological choices
  - For instance, Enterprise Java Beans can handle persistence
- Design principles (identified early in the project's life cycle)
  - Use of patterns
  - Best practices (industry, corporate, project)
  - Reuse strategy

Identifying Design Classes

- An analysis class maps directly to a design class if:
  - It is a simple class
  - It represents a single logical abstraction
    - Typically, entity classes survive relatively intact into design
- A more complex analysis class may:
  - Be split into multiple classes
  - Become a part of another class
  - Become a package
  - Become a subsystem (discussed later)
  - Become a relationship
  - Be partially realized by hardware
  - Not be modeled at all
  - Any combination …

Contracts CO1: makeNewSale()

Operation: makeNewSale()

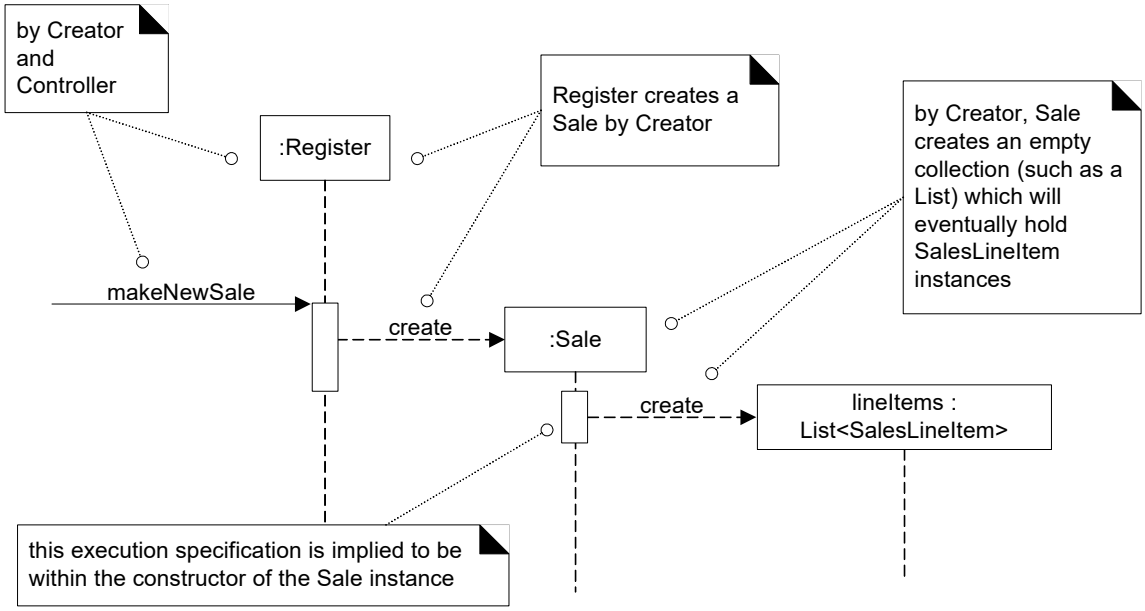
Cross References: Use Cases: Process Sale

Pre-conditions: none

Post-conditions:

- A ***Sale*** instance *s* was created (instance creation).
- *s* was associated with the ***Register*** (association formed).
- Attributes of *s* were initialized.

Creating a New Sale



Contract CO2: enterItem

Operation: enterItem(itemID : ItemID, quantity : integer)

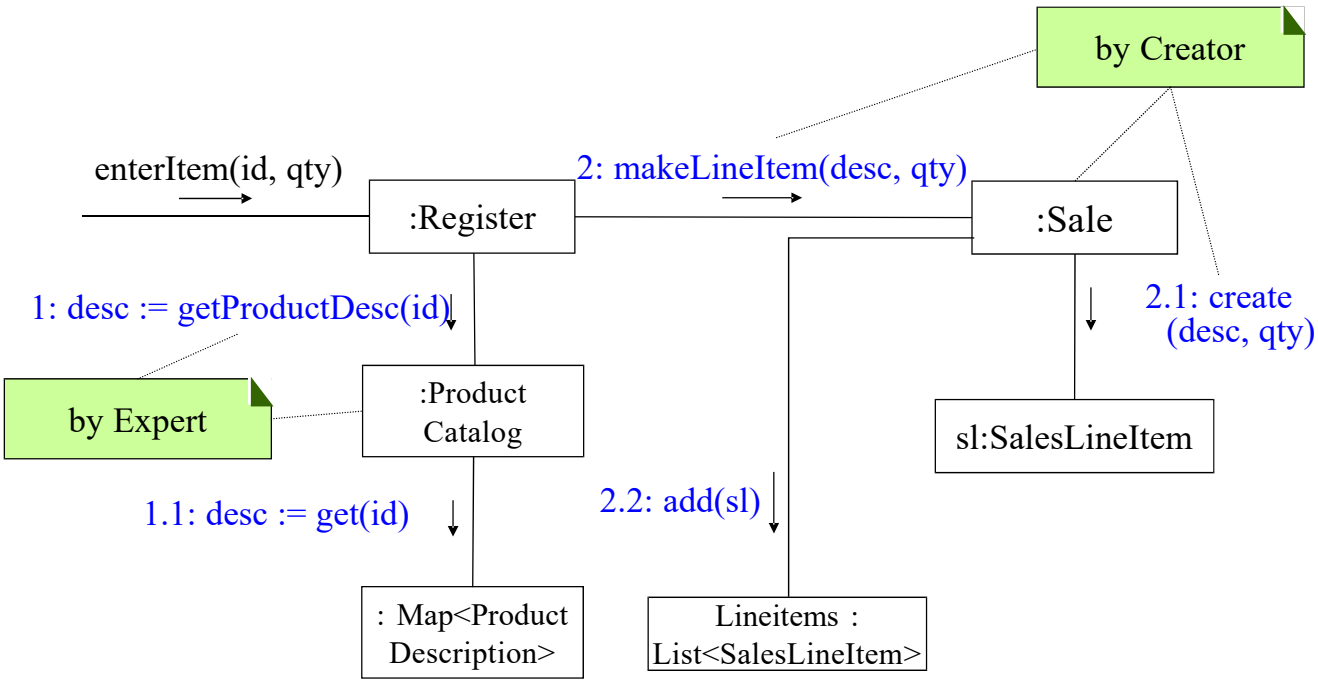
Cross References: Use Cases: Process Sale

Pre-conditions: There is a sale underway.

Post-conditions:

- A ***SalesLineItem*** instance ***sli*** was created (instance creation)
- ***sli*** was associated with the current ***Sale*** (association formed).
- ***sli.quantity*** become quantity (attribute modification).
- ***sli*** was associated with a ***ProductDescription***, based on *itemID* match (association formed).

Communication Diagram: enterItem

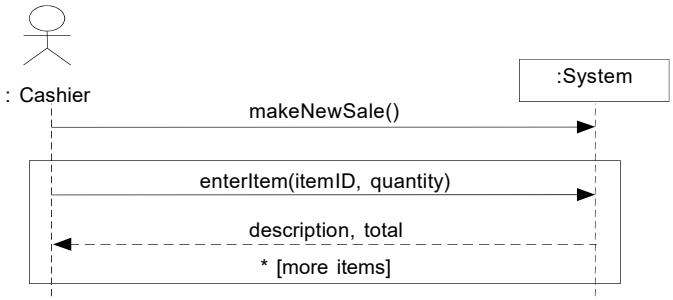


Display item description & price

Simple cash-only *Process Sale* scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.

3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.



- It is not the responsibility of domain objects (such as *Register* or *Sale*) to communicate with the user interface layer ==> *Model-View Separation Pattern*.
- However, the data to be displayed (item *description* and its *price*) should be known at this time.

Model-View Separation

Context/Problem

- It is desirable to de-couple domain objects (i.e., models) from GUIs (i.e., views), to *support increased reuse of domain objects*, and *minimize the impact of changes in the interface* upon the domain objects. What to do?

Solution

- Define the domain (model) classes so that they do not have direct coupling or visibility to the GUI (view) classes, and so that application data and functionality is maintained in domain classes, not GUI classes.

Motivation for MV Separation

- To support cohesive model definitions that focus on the domain processes, rather than on computer interfaces.
- To allow separate development of the model and user interface layers.
- To minimize the impact of requirements changes in the interface upon the domain layer.
- To allow new views to be easily connected to an existing domain layer, without affecting the domain layer.
- To allow multiple simultaneous views on the same model.
- To allow execution of the model layer independent of the GUI layer, such as in a message-processing or batch-mode system.
- To allow easy porting of the model layer to another GUI framework.

Displaying the Sale Total

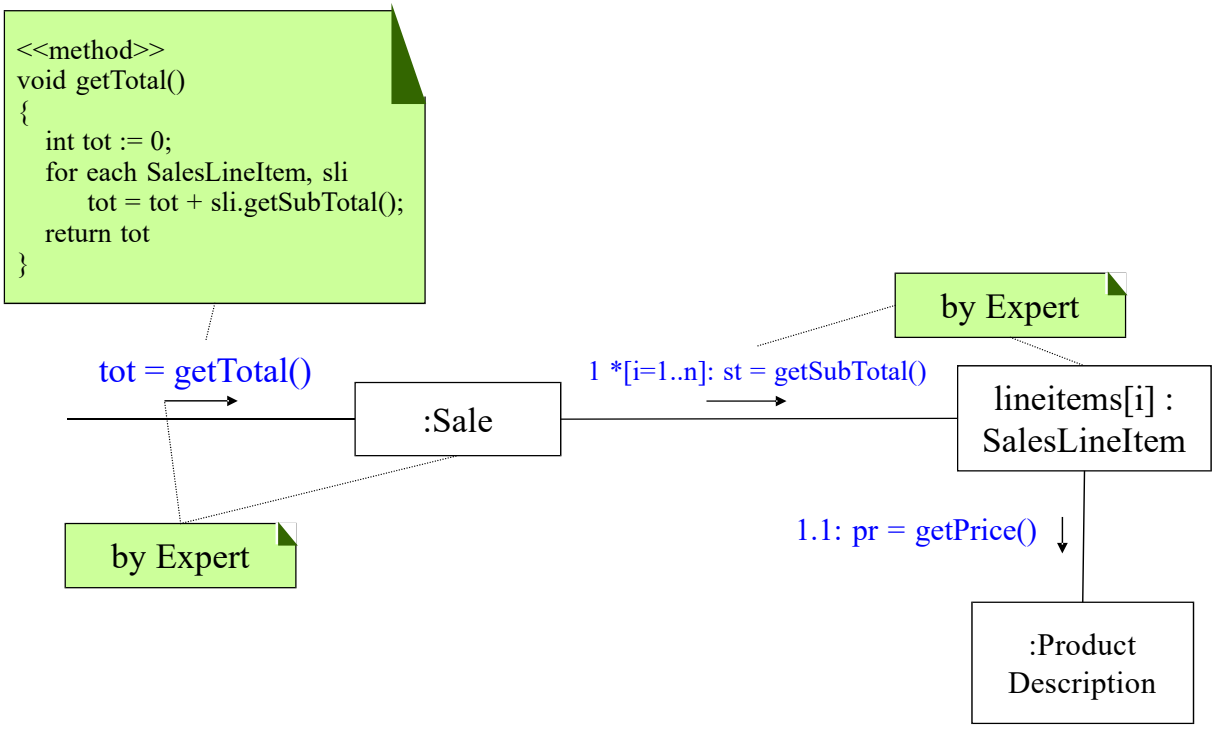
- Remember the Model View Separation Pattern.
- During the creation of collaboration diagrams, do not be concerned with the display of information, *except* insofar as the required information is known.
- Ensure that all information that must be displayed is known and available from the domain objects.

Calculating the Sale Total

1. State the responsibility:
   - Who should be responsible for knowing the sale total?
2. Summarize the information required:
   - The sale total is the sum of the subtotals of all the sales line-items.
   - Sales line-item subtotal := line-item quantity * product description price
3. List the information required to fulfill this responsibility and the classes (i.e., Expert) that know this information.

ProductDescription.price	*ProductDescription*
SalesLineItem.quantity	*SalesLineItem*
all the *SalesLineItems* in the current sale	*Sale*

Communication Diagram: getTotal()



Contract CO3: endSale()

Operation: endSale()

Cross References: Use Cases: Process Sale

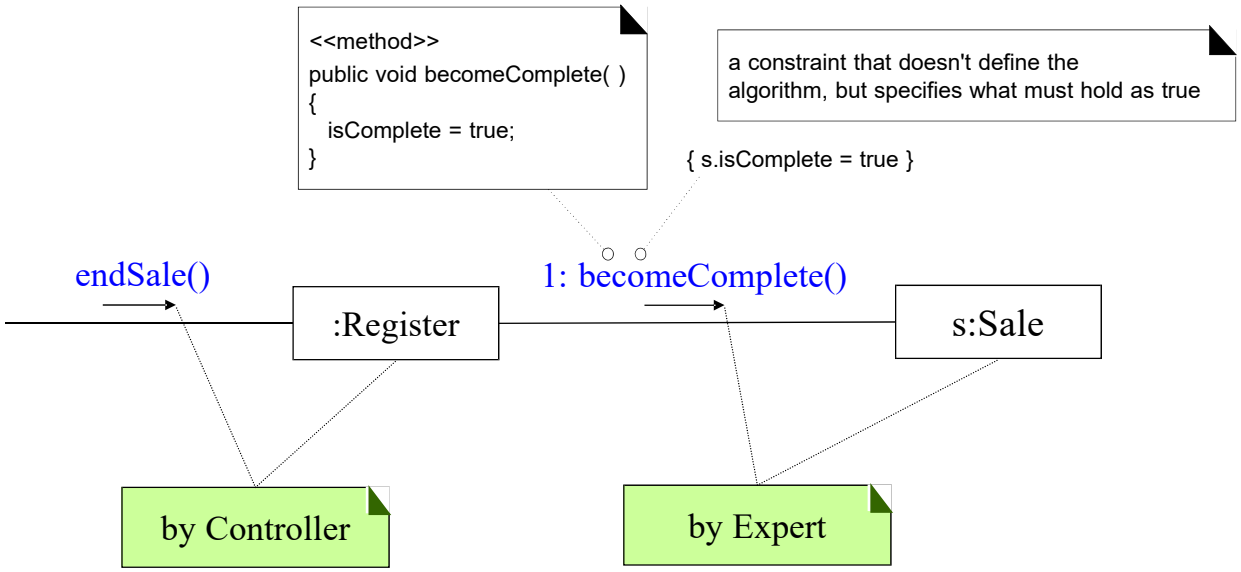
Pre-conditions: There is a sale underway.

Post-conditions:

- ***Sale.isComplete*** become true (attribute modification).

Communication Diagram: endSale()

Constraints, Notes, and Algorithms



Collaboration Diagram: makePayment()

- *Study for yourself!*

Comments on *StartUp* System Operation

- *startUp* operation abstractly represents the initialization phase of execution when an application is launched. It depends on the programming language and OS.
- A common design idiom is to create an ***initial domain object***, which is responsible for creating other domain objects.

```
public class RegisterApplet extends Applet {
   public void  init() {  register = store.getRegister();   }
   …
   private Store store  = new Store();    // initial domain object
   private Register register;
   private Sale sale;
}
```

Interpretation of *startUp*

1. In one collaboration diagram, send a "*<<create>>*" message to create the initial domain object.
2. (optional) If the initial object is taking control of the process, in a second collaboration diagram send a "*run*" message (or something equivalent) to the initial object.

- Create the start up collaboration diagram last. ***Why?***

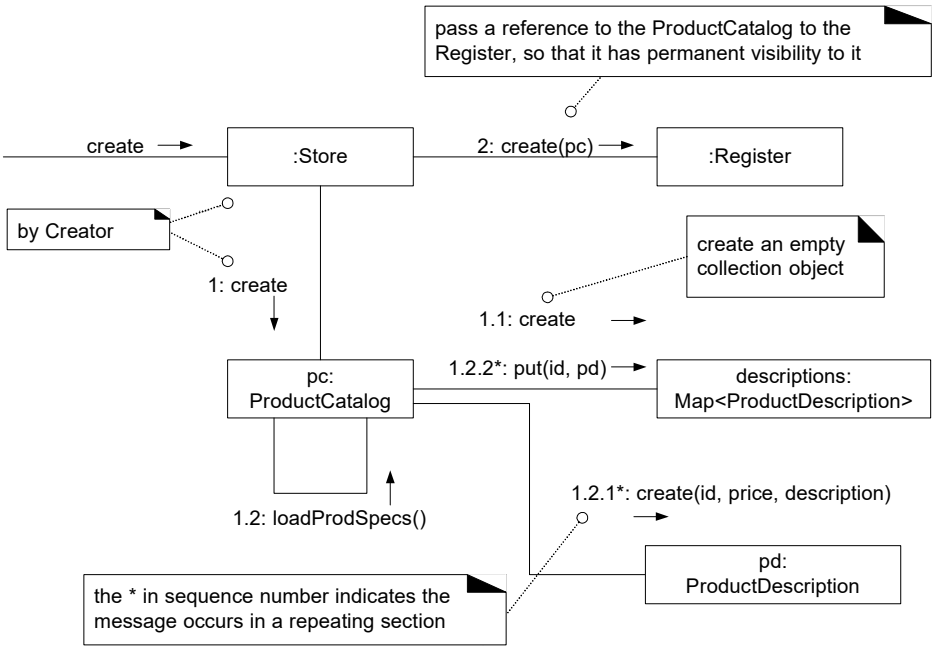
Choosing The Initial Domain Object

- What should be the class of the initial domain object?
- Choose as an initial domain object:
  - A class representing the entire logical information system
    - *Register*
  - A class representing the overall business or organization
    - *Store*
- The choice may be influenced by High cohesion and Low coupling principles.

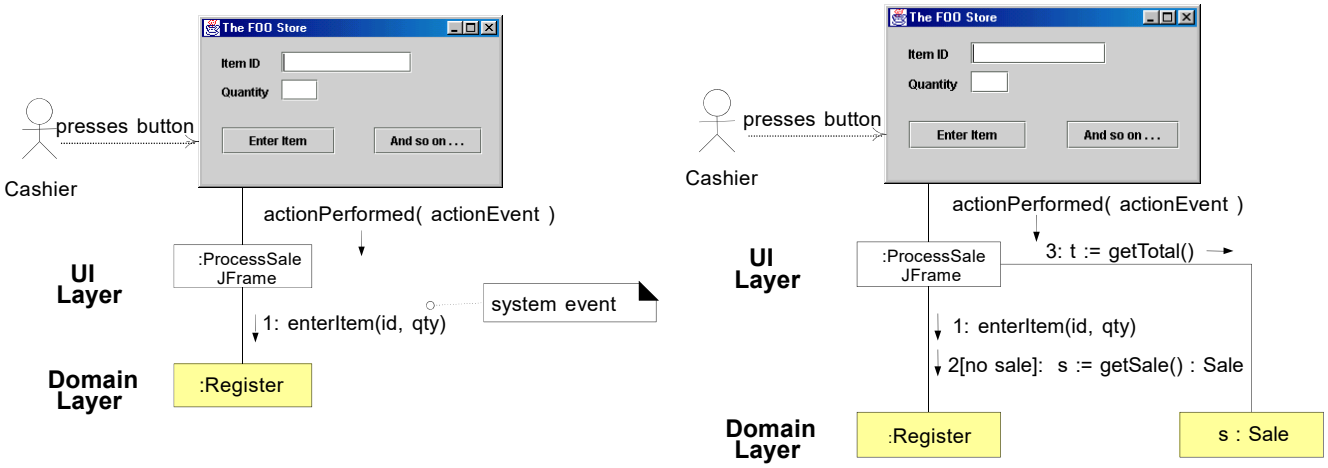
Derived Initialization Tasks From Prior Design Work

- A ***Store***, ***Register***, ***ProductCatalog*** and ***ProductDescription***s needs to be created.
- The ***ProductCatalog*** needs to be associated with ***ProductDescription***s.
- ***Store*** needs to be associated with ***ProductCatalog***.
- ***Store*** needs to be associated with ***Register***.
- ***Register*** needs to be associated with ***ProductCatalog***.

Collaboration Diagram: startUp()



Connecting UI Layer to Domain Layer



Determining Visibility & Creating Class Diagrams

Objectives

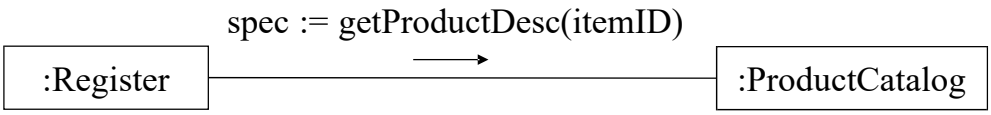
Identify four kinds of visibility

Design to establish visibility

Understand the interplay between interaction & class diagrams

Visibility

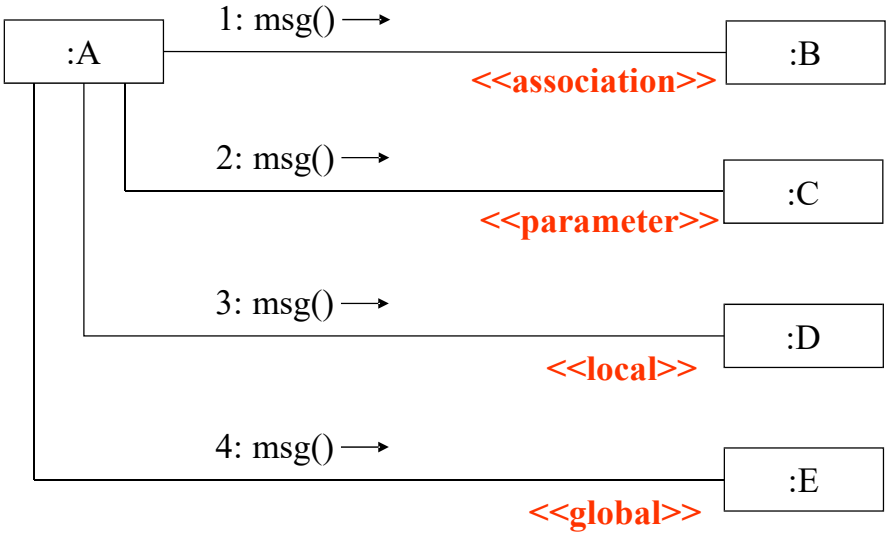
- For an object *A* (sender) to send a message to an object *B* (receiver), *B* must be *visible* to *A*.
- Visibility is the ability of one object to "see" or have reference to another.



Visibility From A to B

- Attribute visibility
  - *B* is an attribute of *A*.
- Parameter visibility
  - *B* is a parameter of a method of *A*.
- Local visibility
  - *B* is declared as a (non-parameter) local object in a method of *A*.
- Global visibility
  - *B* is in some way globally visible.

UML Stereotypes for Visibility



Activities and Dependencies

The creation of design class diagram (DCD) relies upon

- *Interaction diagram* -- from this, the designer identifies the software classes that participate in the solution, plus the methods of the classes.
- *Domain model* -- from this, the designer adds detail (such as attributes) to the class definitions.

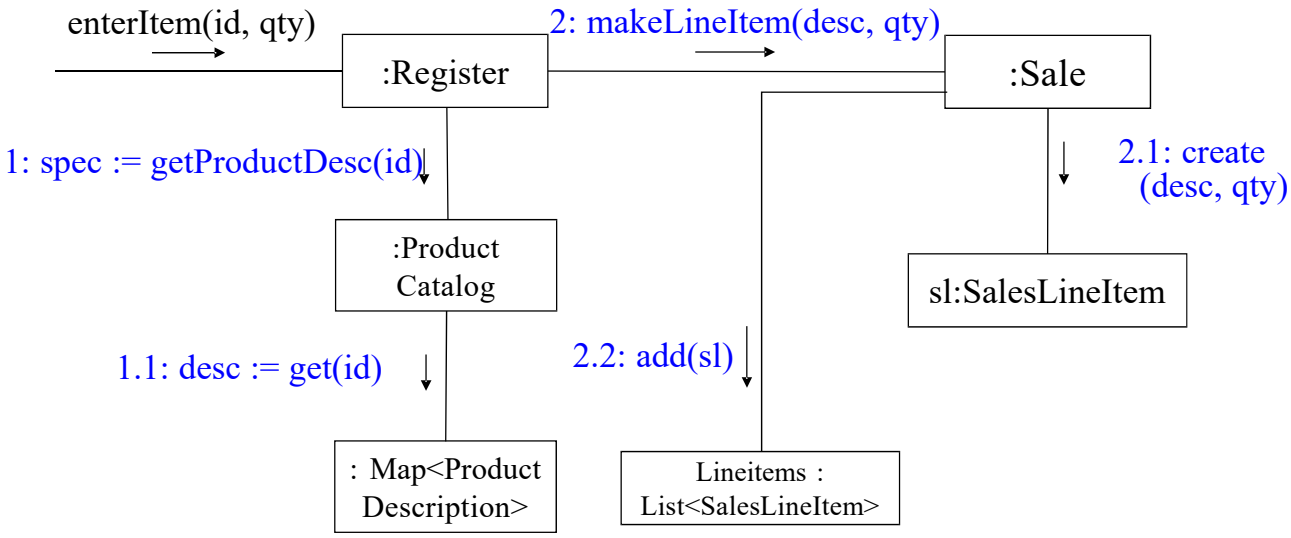
Interplay Between Interaction and Class Diagrams

- In practice, interaction diagrams (dynamic model) and class diagrams (static model) are usually created in parallel.
- Draft class diagrams can be sketched early in the design phase prior to the creation of interaction diagrams ==> can also be used as an alternative to CRC cards.
- The designer must iterate between the two kinds of models, driving them to converge on acceptable solution.

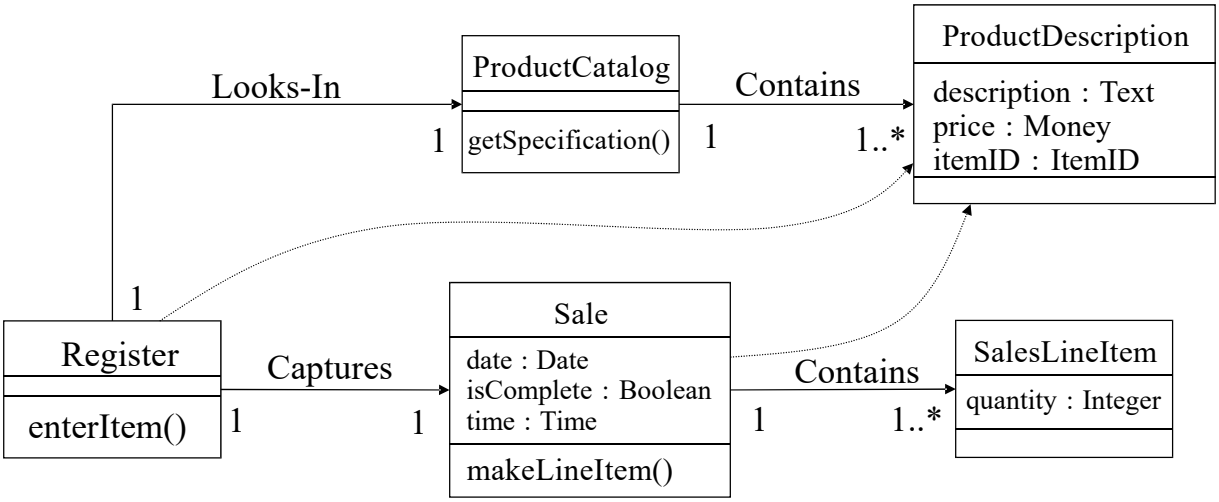
How to Create Design Class Diagram

1. Identify and draw all the classes participating in the software solution by analyzing the interaction diagram.
2. Duplicate the attributes from the associated concepts in the domain model.
3. Add method names by analyzing the interaction diagram.
4. Add type information to the attributes and methods.
5. Add the associations necessary to support the required attribute visibility.
6. Add navigability arrows to the associations to indicate the direction of attribute visibility.
7. Add dependency relationship lines to indicate non-attribute visibility.

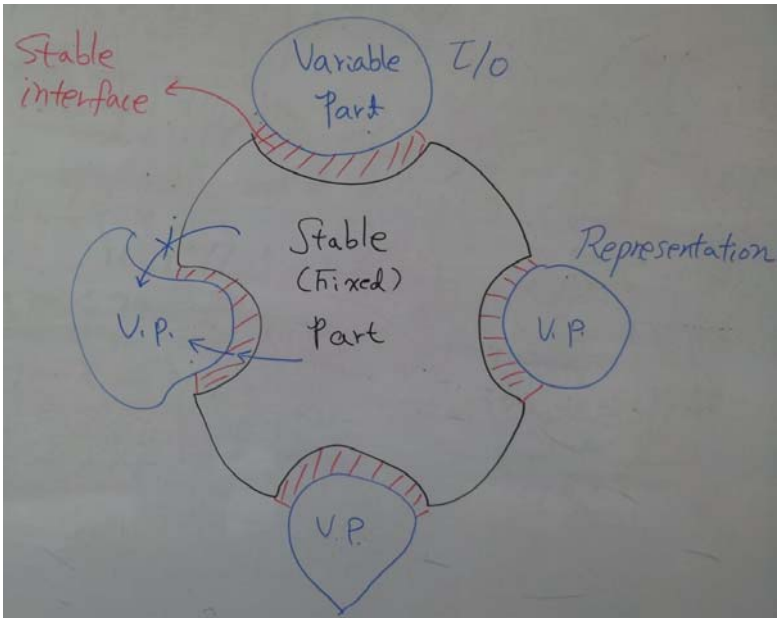
Communication Diagram: enterItem



Derived Design Class Diagram



GRASP Pattern (II)



GRASP: More Patterns For Assigning Responsibilities

Objectives

Learn to apply the remaining GRASP patterns

GRASP Patterns

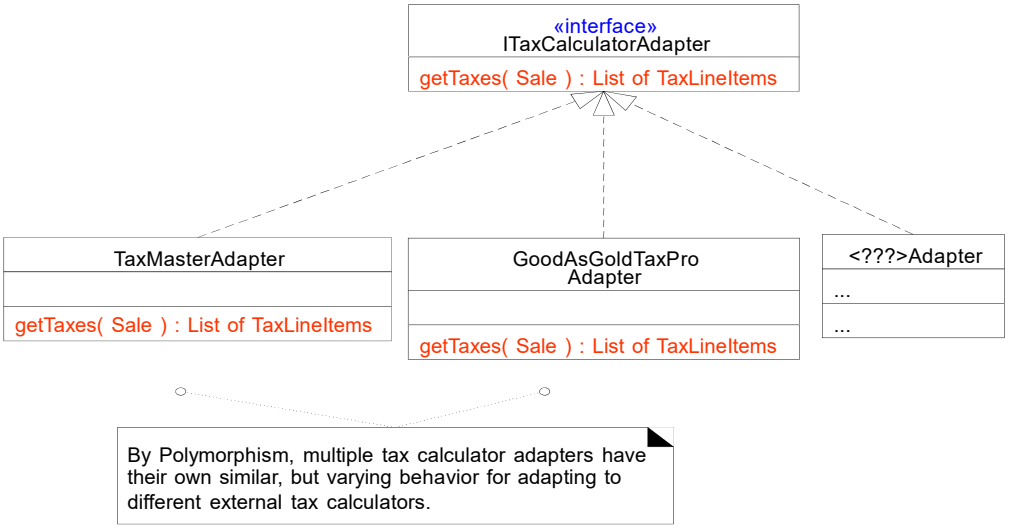
- Information Expert (or Expert)
- Creator
- High Cohesion
- Low Coupling
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Polymorphism

- Problem
  - How to handle alternatives based on type? How do you create pluggable software components?
- Solution
  - **When related alternatives or behaviors vary by type (class), assign responsibility for the behavior -- using polymorphic operations -- to the types for which the behavior varies.**
  - **Corollary: Do not test for the type of an object and use conditional logic.**

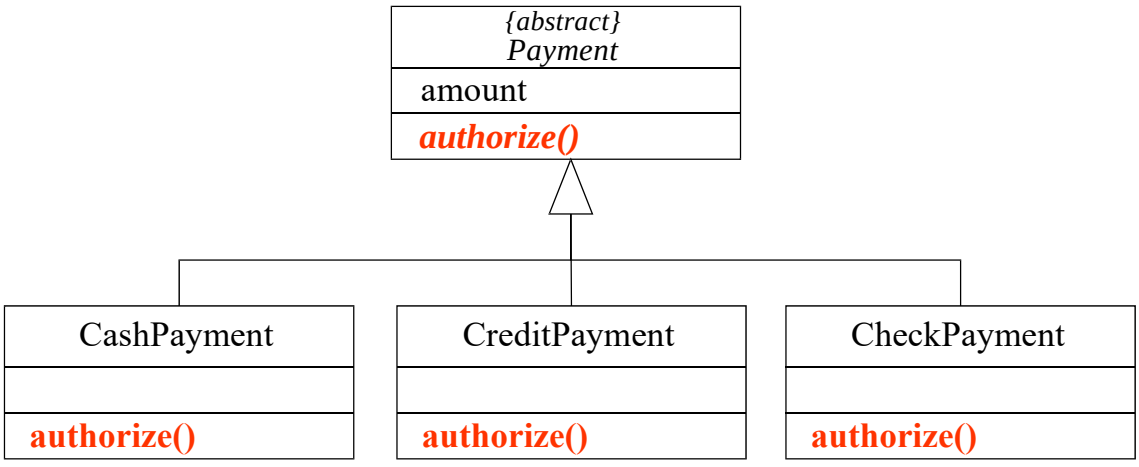
Example: Polymorphism

- In the NextGen POS application, who should be responsible for handling the varying external tax calculator interfaces (like TCP sockets, SOAP, or RMI)?



Example: Polymorphism

- In the NextGen POS application, who should be responsible for authorizing different kinds of payments?

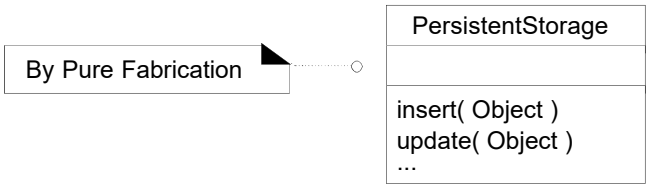


Pure Fabrication

- Problem
  - What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling or other goals but solutions offered by Expert (for example) are note appropriate?
    - There are situations in which assigning responsibilities only to domain classes leads to problems in terms of poor cohesion and coupling, or low reuse potential.
- Solution
  - **Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent anything in the problem domain -- something made up (i.e., *fabrication*).**

Example: Pure Fabrication

- Suppose that support is needed to save *Sale* instances in a relational database. Who should do that?
- By Information Expert, there is some justification to assign this responsibility to the *Sale* class itself, but …



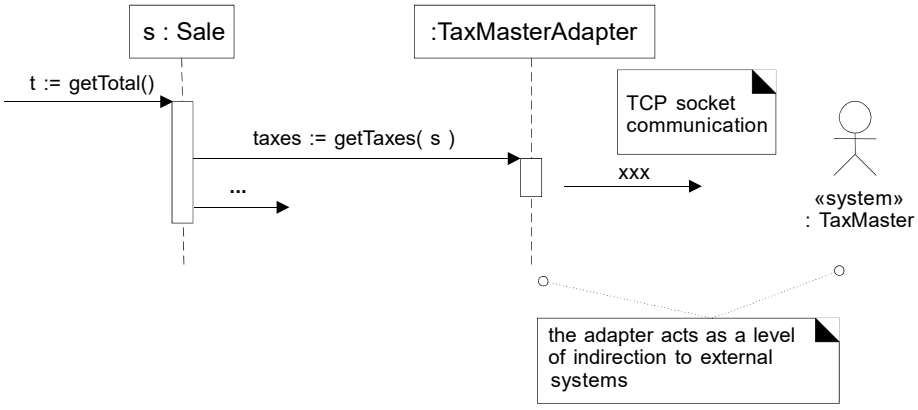
- Note: Pure Fabrications are usually a consequence of *behavior decomposition* rather than *representational decomposition*. That is, they are a kind of function-centric or behavioral objects. ***Caution***: *Do not overuse!*

Indirection

- Problem
  - Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to de-couple objects so that Low Coupling is supported and reuse potential remains higher?
- Solution
  - **Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. The intermediary creates an *indirection* between the other components or services.**

Example: Indirection

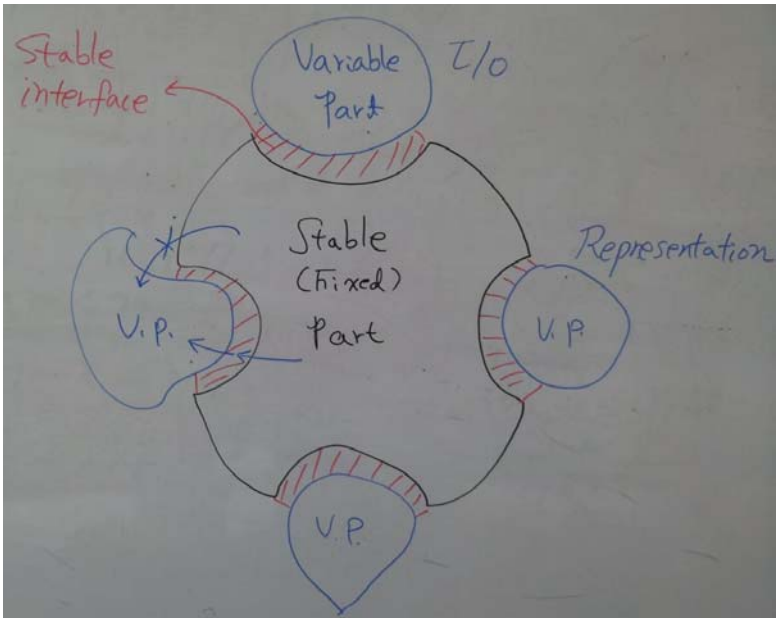
- *TaxCalculatorAdapter* objects acts as intermediaries to the external tax calculators. By adding a level of indirection and adding polymorphism, the adapter objects protect the inner design against variations in the external interfaces.



- Decoupling *Sale* and the relational database through *PersistentStorage* is also an example of Indirection.

Protected Variations

- Problem
  - How to assign objects, subsystems, and systems so that the ***variations or instability*** in these elements does ***not have an undesirable impact on other elements***?



- Solution
  - **Identify points of predicted variations or instability; assign responsibilities to create a stable interface around them.**

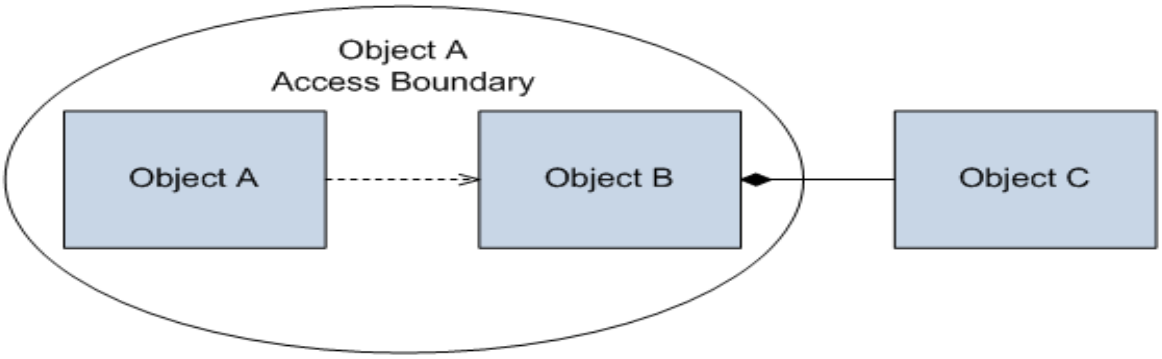
Mechanism Motivated by PV

- Core Protected Variations Mechanisms
  - Data encapsulation, interfaces, polymorphism, brokers, virtual machines, etc.
- Data-Driven Designs
- Service Lookup
- Interpreter-Driven Design
- Reflective or Meta-level Designs
- Uniform access
- The Liskov Substitution Principle (LSP)
- Structure-Hiding Designs (Don't Talk to Strangers)

Don't Talk to Strangers

- Problem
  - Who, to avoid knowing about the structure of indirect objects?
  - If an object has knowledge of the internal structure of other objects, it suffers from high coupling.
  - If a client has to use a service or obtain information from an indirect object, how can it do so without being coupled to knowledge of the internal structure of its direct server or indirect objects?

Don't Talk to Strangers



- Solution
  - **Assign the responsibility to a client's direct object (i.e., *familiar*) to collaborate with an indirect object (i.e., *stranger*), so that the client does not know about the indirect object.**

Law of Demeter
(Principle of Least Knowledge)

Constrains on what objects you should send a message to within a method.

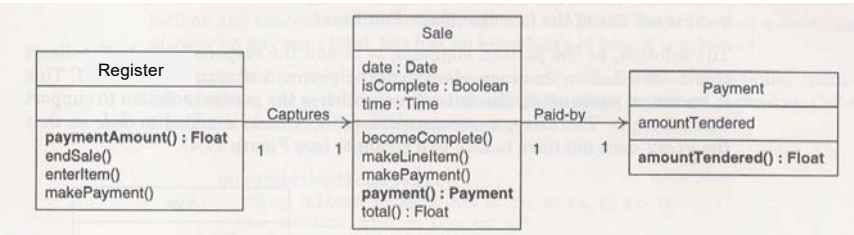
- The *this* object (or *self*).
- The parameter of the method.
- An attribute of *self*.
- An element of a collection which is an attribute of *self*.
- An object created within the method.

Example: Don't Talk ...

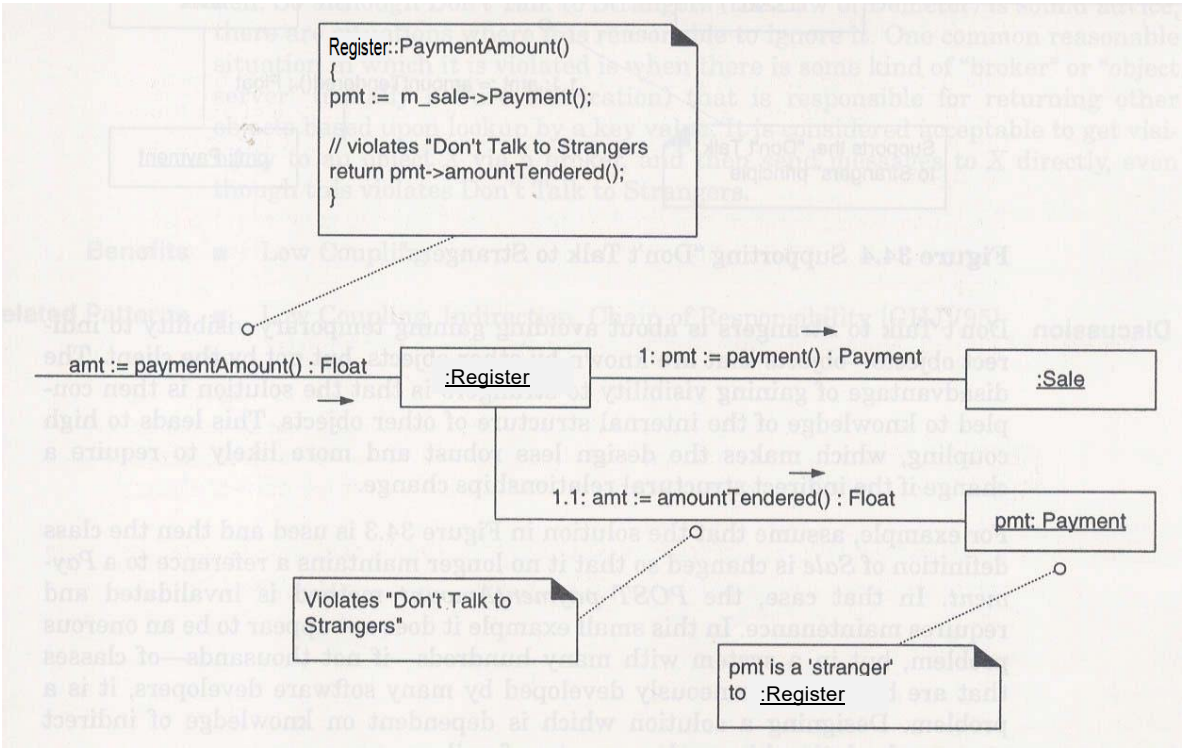
Assume that:

- *Register* instance has an attribute referring to a *Sale*, which has an attribute referring to a *Payment*.
- *Register* supports the *paymentAmount()* operation, which returns the current amount tendered.
- *Sale* supports *payment()* operation, which returns the *Payment* instance associated with the *Sale*.

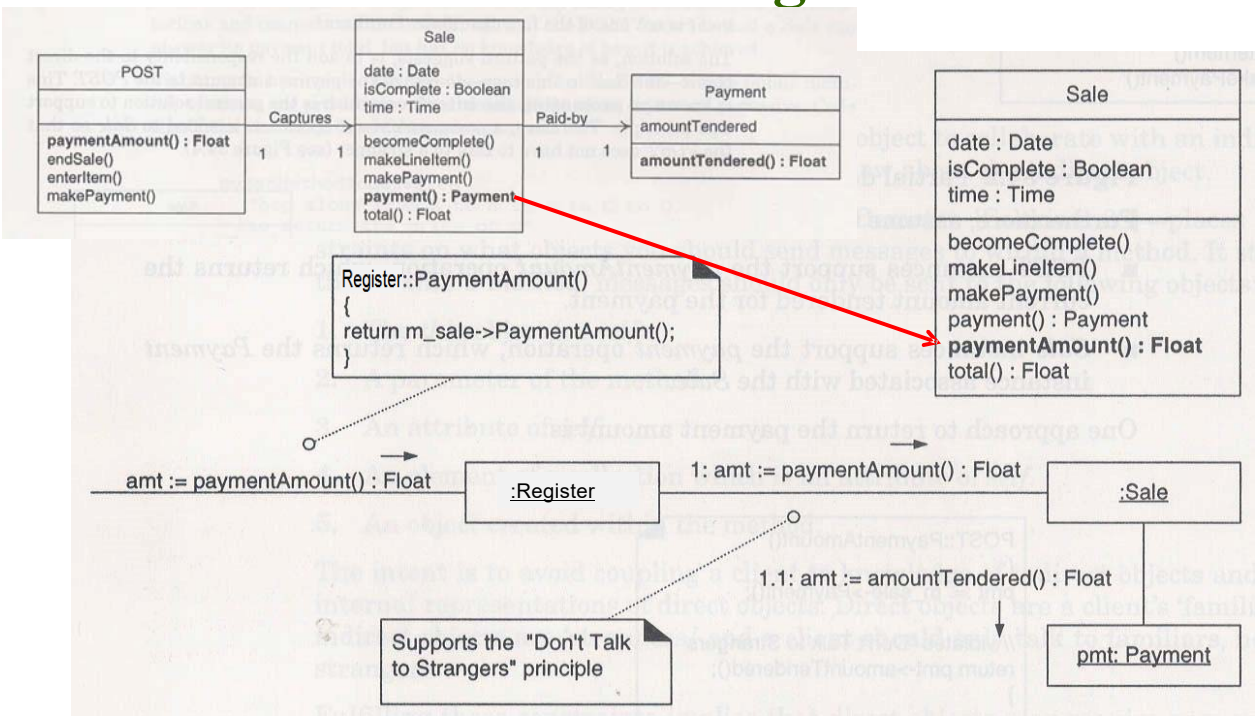
  Then, how should the *Register* instance return the payment amount?



A Solution Violating Don't Talk …

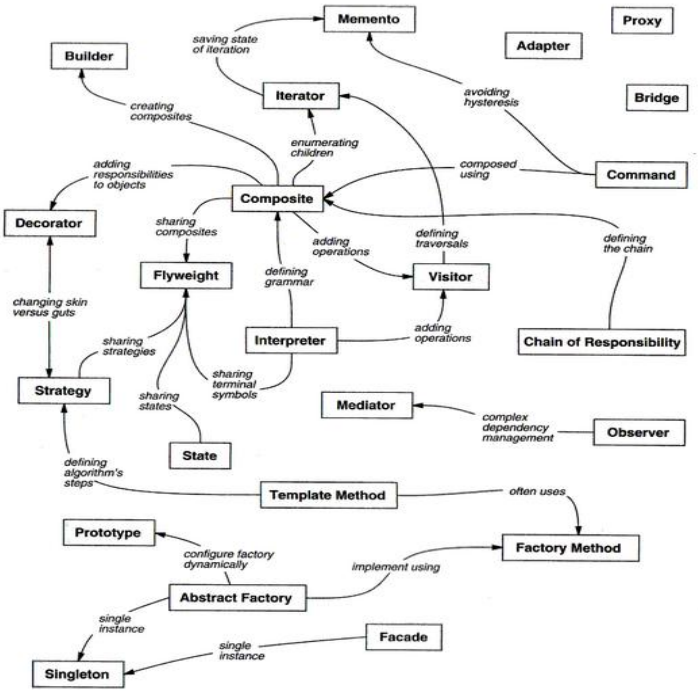


A Solution Advocating Don't Talk …



Object-Oriented Analysis and Design using UML and Patterns

More Object Design with GoF



More Object Design with GoF

Objectives

Apply some GoF design patterns

Show GRASP principles as a generalization of other design patterns

Adapter Pattern

Problem

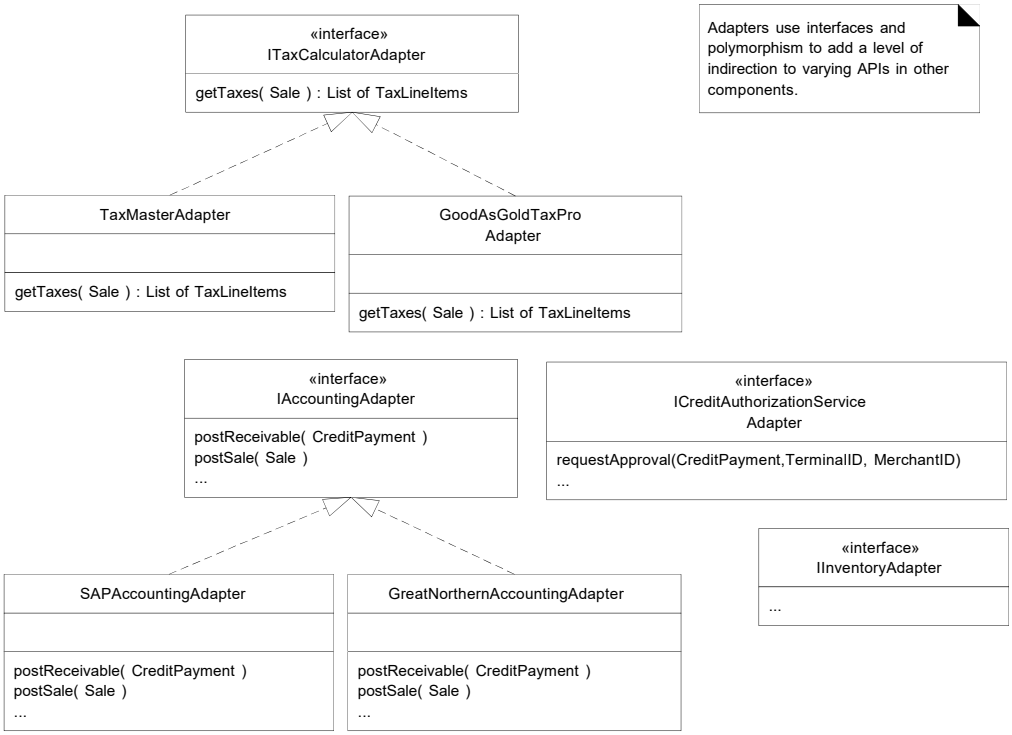
How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

Solution

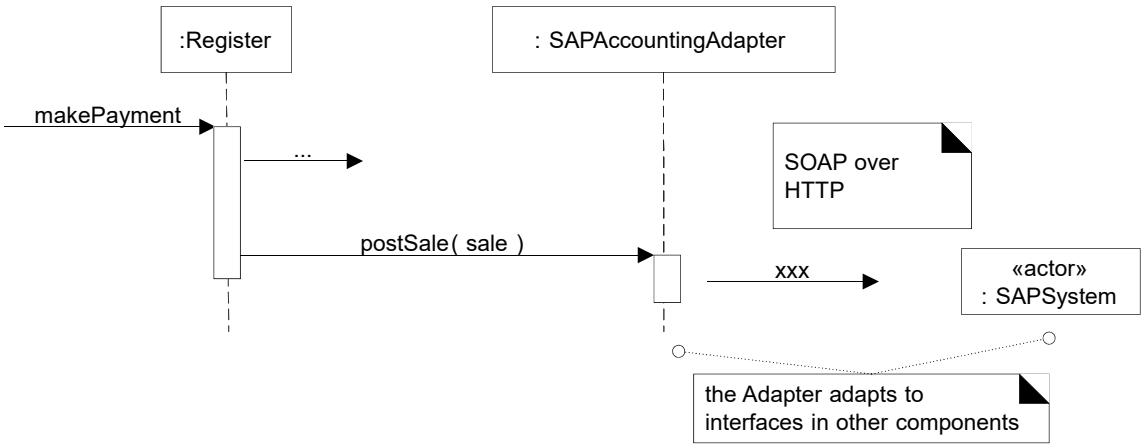
Convert the original interface of a component into another interface, through an intermediate adapter object

The NextGen POS system needs to support several kinds of external third-party services, including tax calculators, credit authorization services, inventory systems, and accounting systems, among others. Each has a different API, which can't be changed.

Adapter Pattern (Cont'd)



Adapter Pattern (Cont'd)



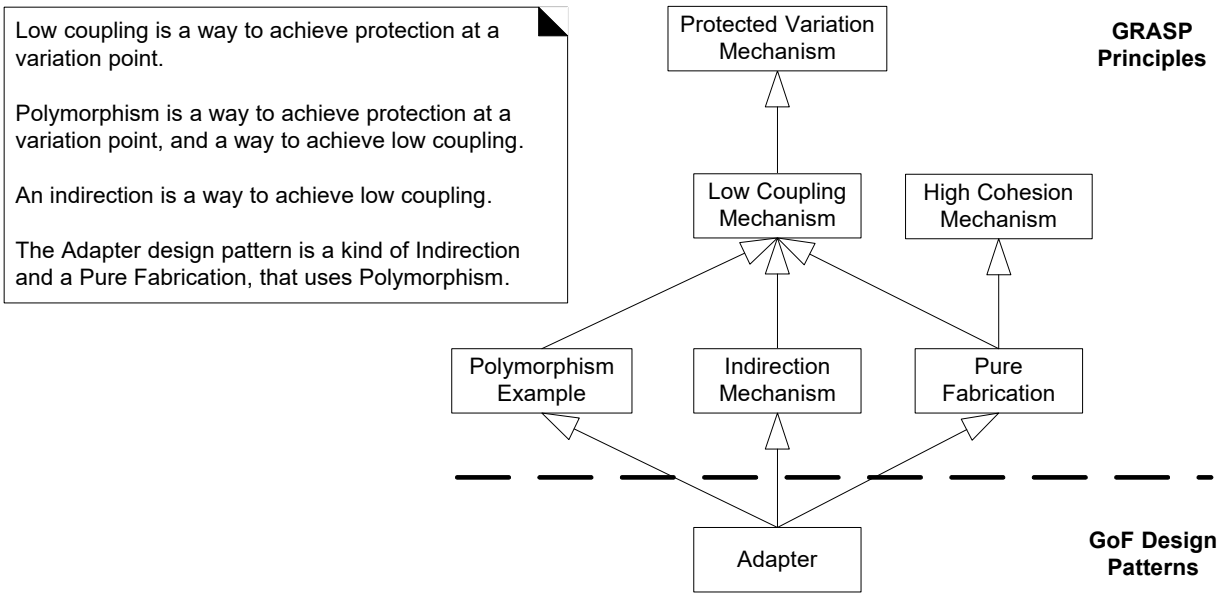
Adapter Pattern vs. GRASP Pattern

Adapter supports *Protected Variations* with respect to changing external interfaces or third-party packages through the use of an *Indirection* that applies interfaces and *Polymorphism*

The published patterns (e.g., Pattern Almanac) lists around +1000 patterns
➔ Pattern Overload!

A Solution: See the underlying principles

Adapter Pattern vs. GRASP Pattern (Cont'd)



Note: Factory is not GoF pattern

Factory Pattern

Problem

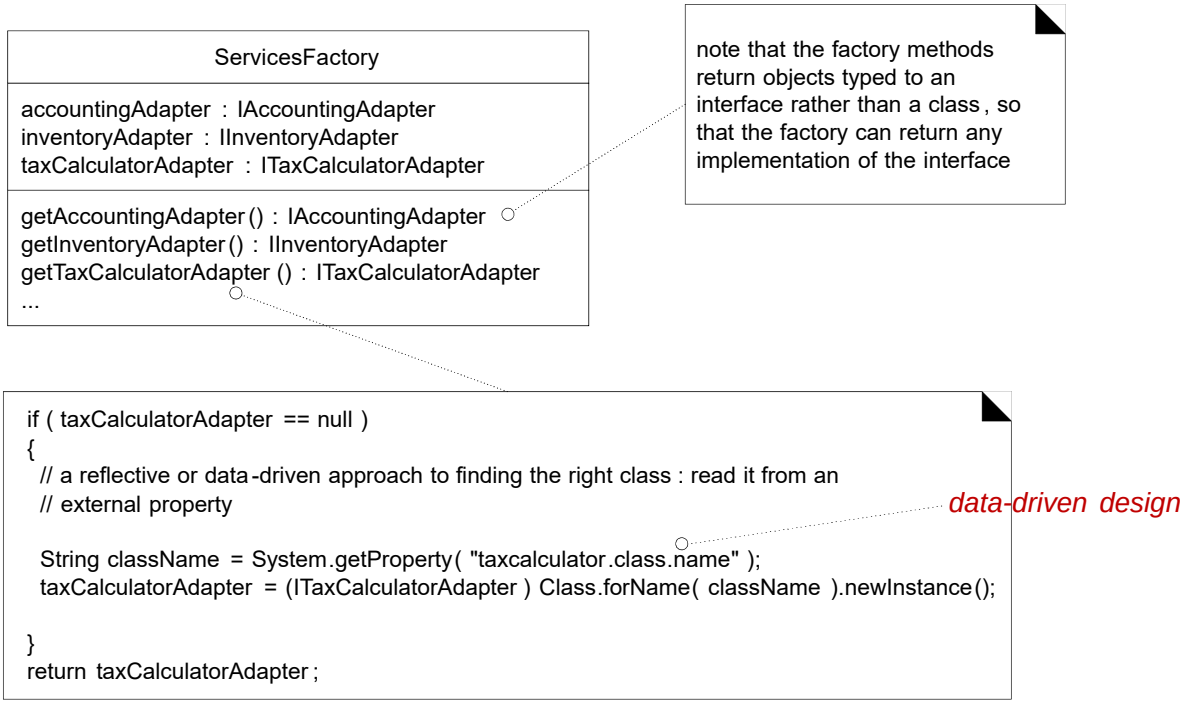
Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?

Solution

Create a **Pure Fabrication** object called a **Factory** that handles the creation

The adapter raises a new problem in the design: In the prior Adapter pattern solution for external services with varying interfaces, who creates the adapters? And how to determine which class of adapter to create, such as *TaxMaster-Adapter or GoodAsGoldTaxProAdapter*?

Factory Pattern (Cont'd)



Advantages of Factory Pattern

Separate the responsibility of complex creation into cohesive helper objects

Hide potentially complex creation logic

Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling

Singleton Pattern

Problem

Exactly one instance of a class is allowed. it is a "singleton." Objects need a global and single point of access

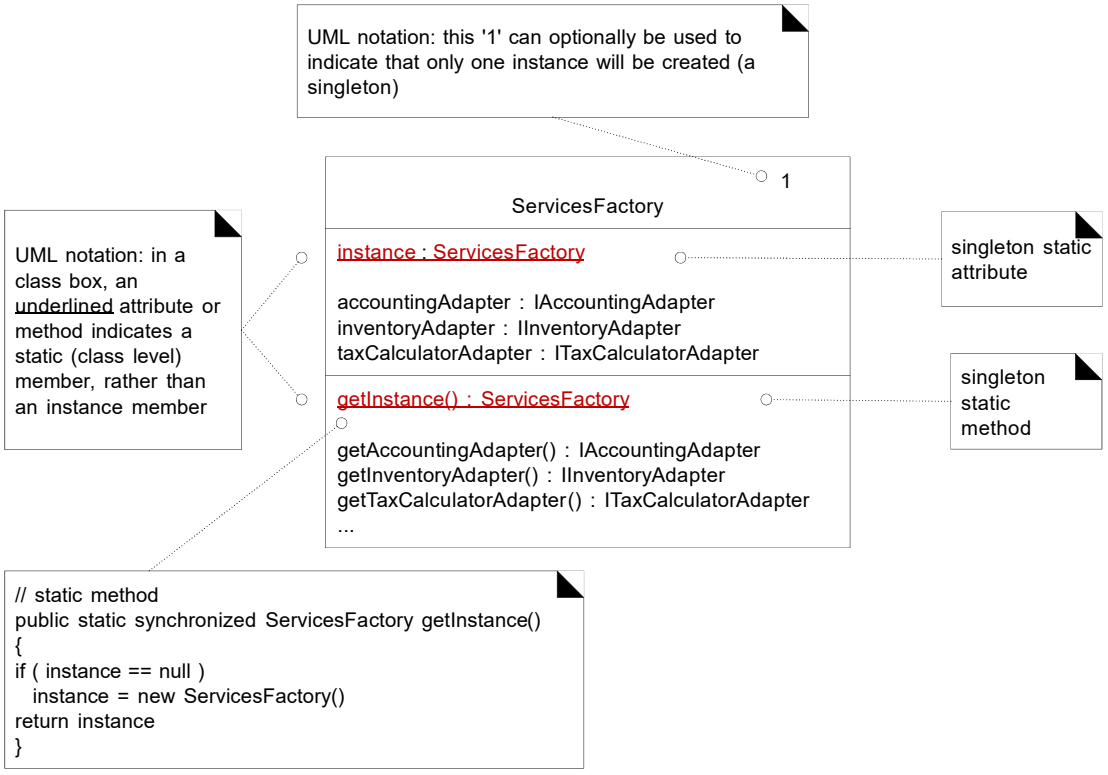
Solution

Define a static method of the class that returns the singleton

The *ServicesFactory* raises another new problem in the design: Who creates the factory itself, and how is it accessed?

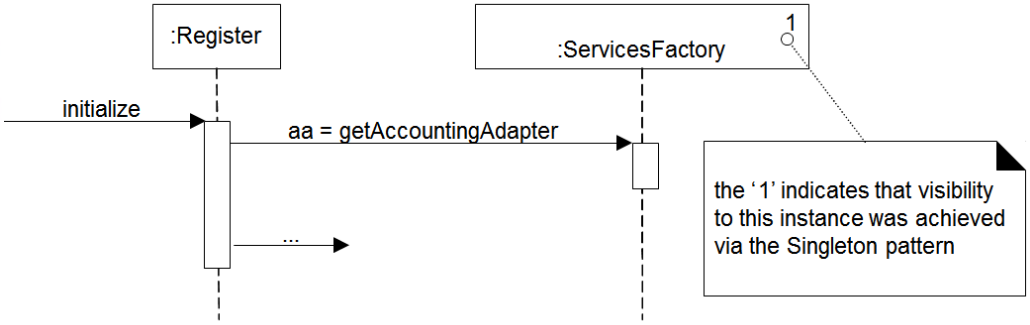
First, observe that only one instance of the factory is needed within the process. Second, quick reflection suggests that the methods of this factory may need to be called from various places in the code, as different places need access to the adapters for calling on the external services. Thus, there is a visibility problem: How to get visibility to this single *ServicesFactory* instance?

Singleton Pattern (Cont'd)

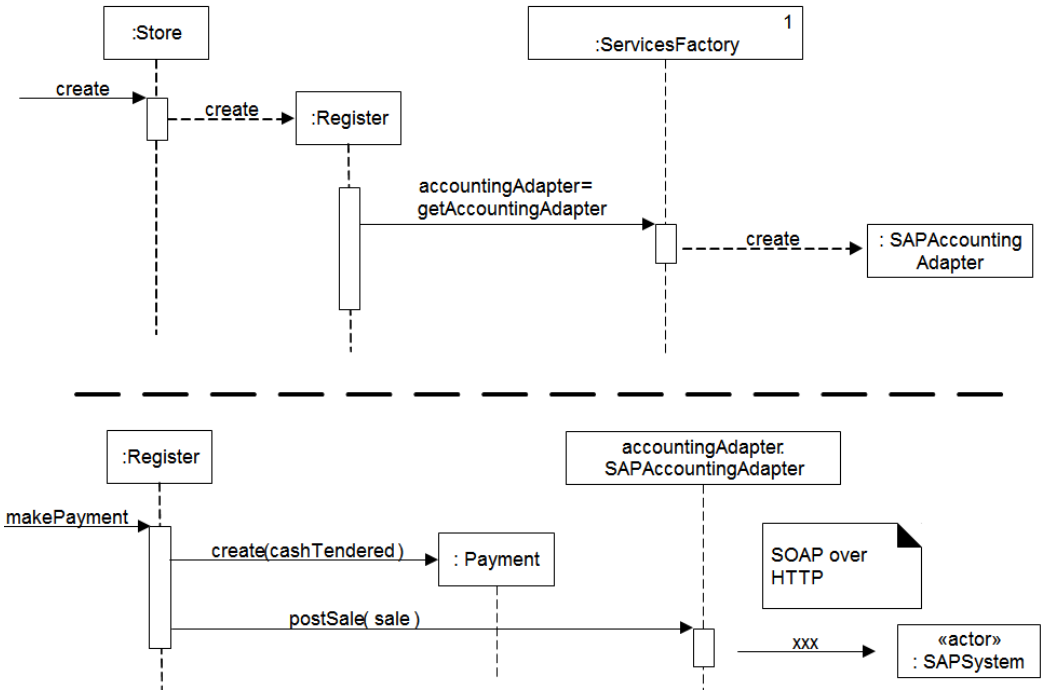


Singleton Pattern (Cont'd)

```
public class Register
{

public void initialize()
{
   … do some work …

   // accessing the singleton Factory via the getInstance call
   accountingAdapter =
      ServicesFactory.getInstance().getAccountingAdapter();

   … do some work …
}

// other methods…

} // end of class
```



A combination of Adapter, Factory, and Singleton patterns have been used to provide Protected Variations from the varying interfaces of external tax calculators, accounting systems, and so forth



Strategy Pattern

Problem

How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies?

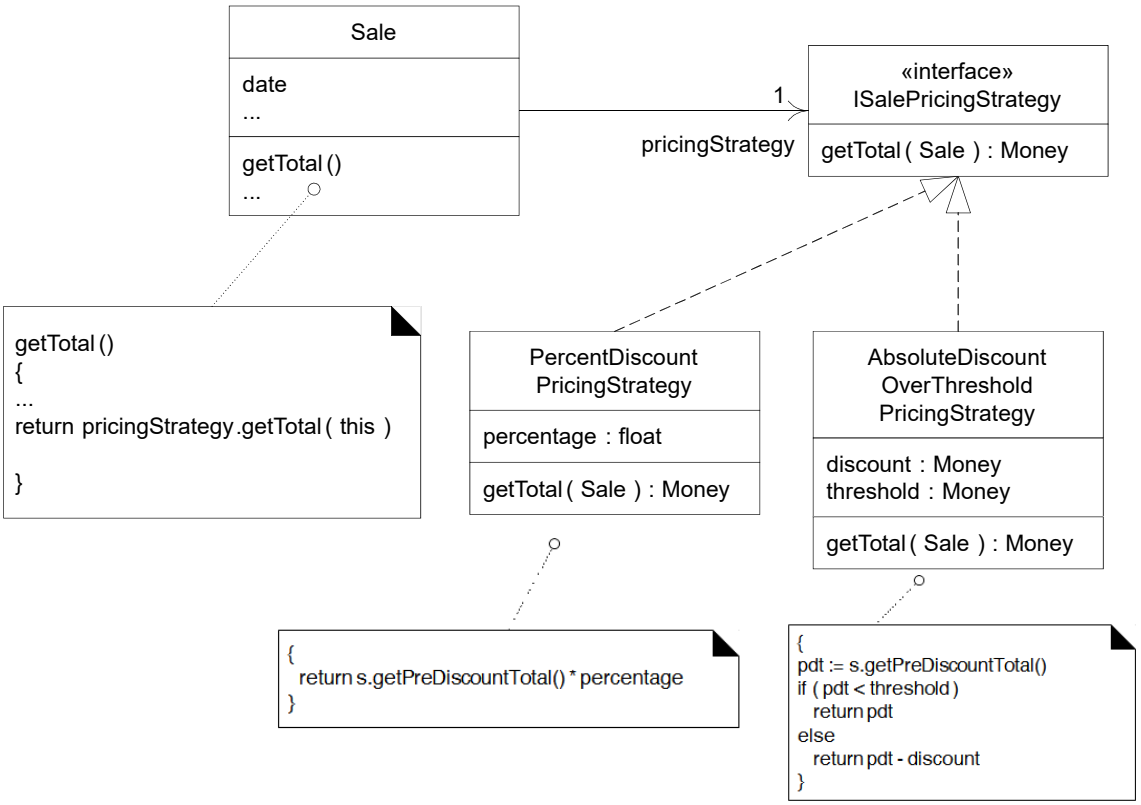
Solution

Define each algorithm/policy/strategy in a separate class, with a common interface

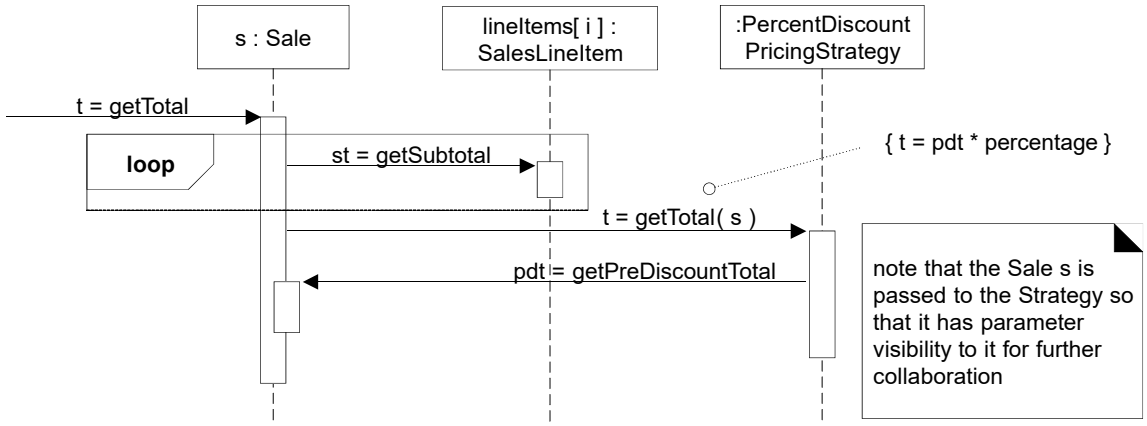
The next design problem to be resolved is to provide more complex pricing logic, such as a storewide discount for the day, senior citizen discounts, and so forth

The pricing strategy (which may also be called a rule, policy, or algorithm) for a sale can vary. During one period it may be 10% off all sales, later it may be $10 off if the sale total is greater than $200, and myriad other variations. How do we design for these varying pricing algorithms?

Strategy Pattern (Cont'd)



Strategy Pattern (Cont'd)



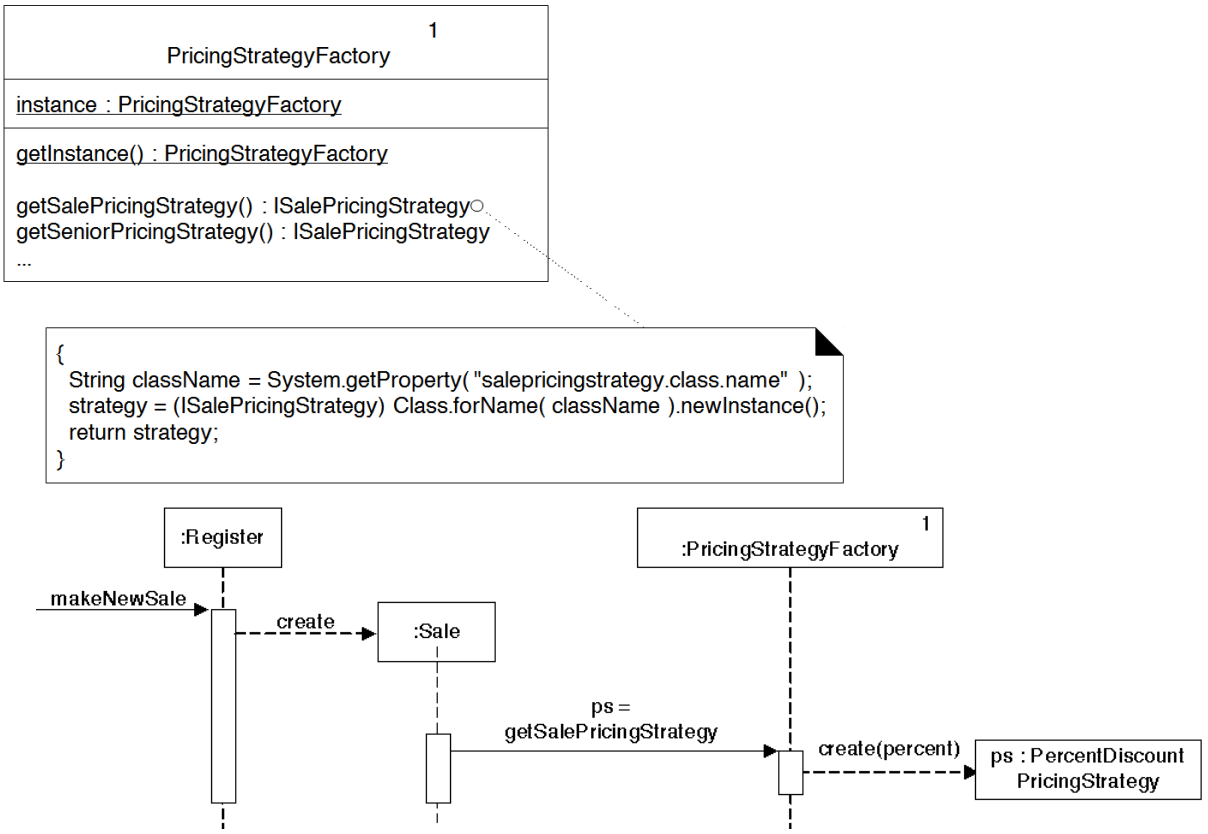
Creating a Strategy with a Factory

There are different pricing algorithms or strategies, and they change over time. Who should create the strategy?

Why separate Factory other than *ServicesFactory*?

Note that because of the frequently changing pricing policy (it could be every hour), it is *not* desirable to cache the created strategy instance in a field of the *PricingStrategyFactory*, but rather to re-create one each time, by reading the external property for its class name, and then instantiating the strategy.

Creating a Strategy with a Factory (Cont'd)



More on Pricing Strategies

How do we handle the case of multiple, conflicting pricing policies? For example, suppose a store has the following policies in effect today (Monday):

- 20% senior discount policy
- preferred customer discount of 15% off sales over $400
- on Monday, there is $50 off purchases over $500
- buy 1 case of Darjeeling tea, get 15% discount off of everything

Suppose a senior who is also a preferred customer buys 1 case of Darjeeling tea, and $600 of veggie burgers (clearly an enthusiastic vegetarian who loves chai). What pricing policy should be applied?

To clarify: There are now pricing strategies that attach to the sale by virtue of three factors:

1. time period (Monday)
2. customer type (senior)
3. a particular line item product (Darjeeling tea)

More on Pricing Strategies (Cont'd)

There can exist multiple co-existing strategies

Pricing strategy can be related to the type of customer

Design implications: The customer type must be known by the ***StrategyFactory*** at the time of creation of a pricing strategy for the customer.

Similarly, a pricing strategy can be related to the type of product being bought

Design implications: The ***ProductDescription*** must be known by the ***StrategyFactory*** at the time of creation of a pricing strategy influenced by the product.

Is there a way to change the design so that the ***Sale*** object does not know if it is dealing with one or many pricing strategies, and also offer a design for the conflict resolution?

Composite Pattern

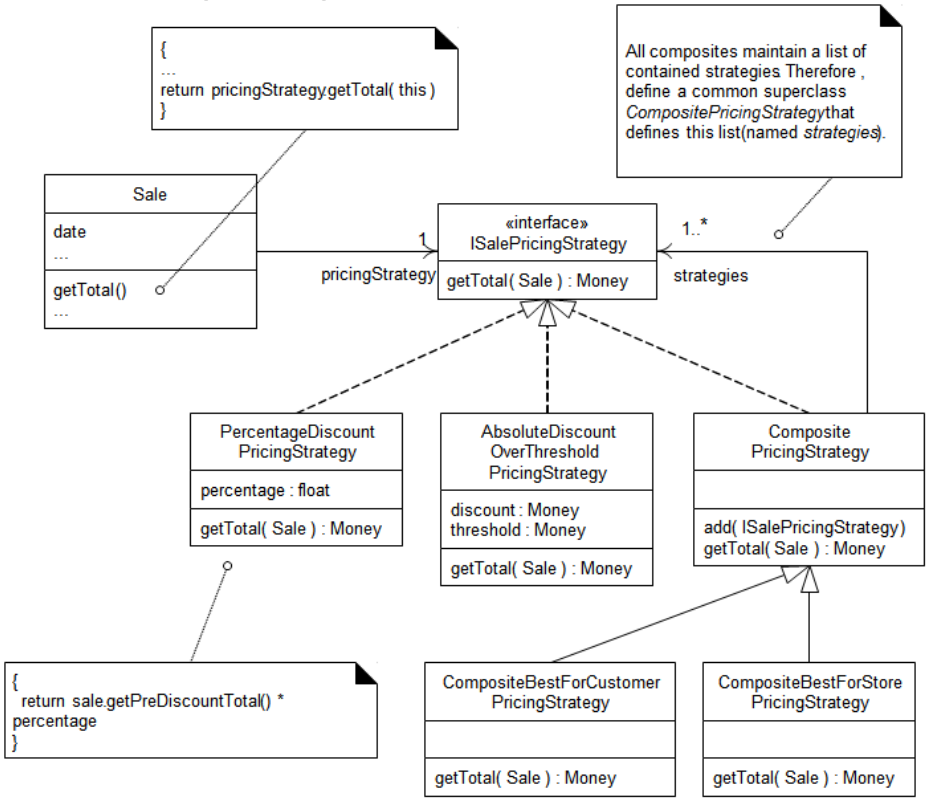
Problem

How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?

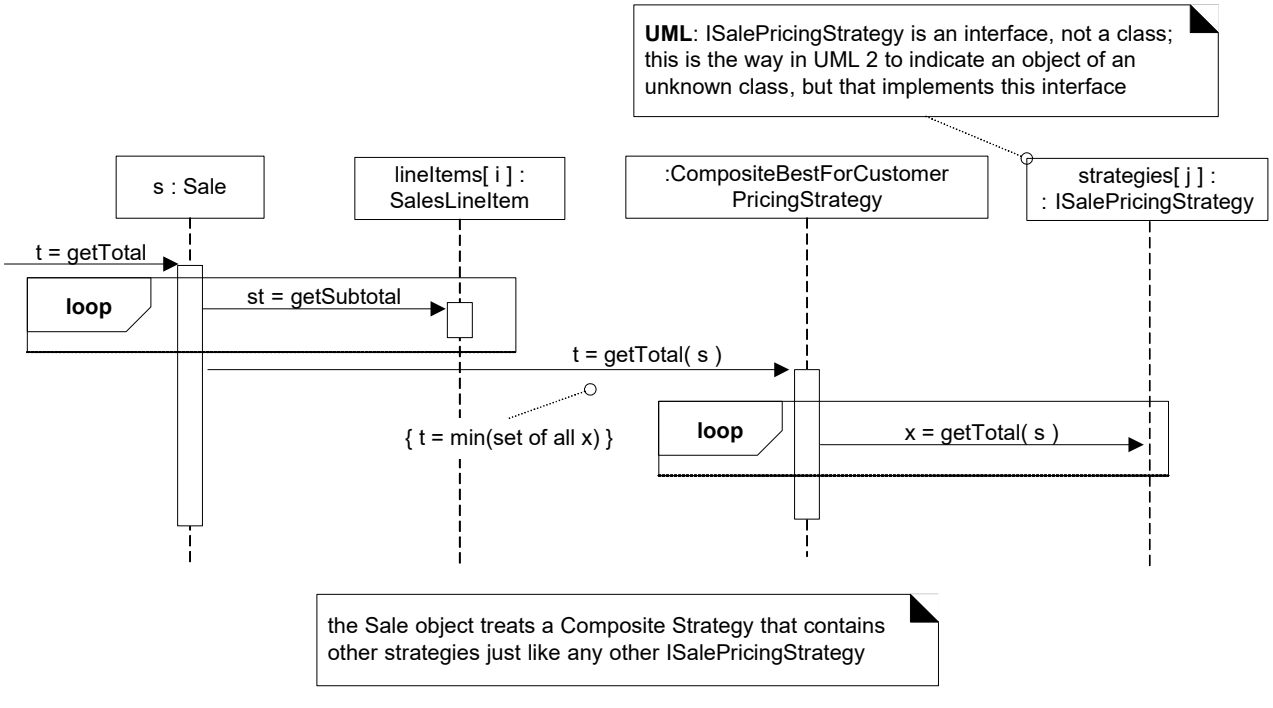
Solution

Define classes for composite and atomic objects so that they implement the same interface

Composite Pattern (Cont'd)



Composite Pattern (Cont'd)

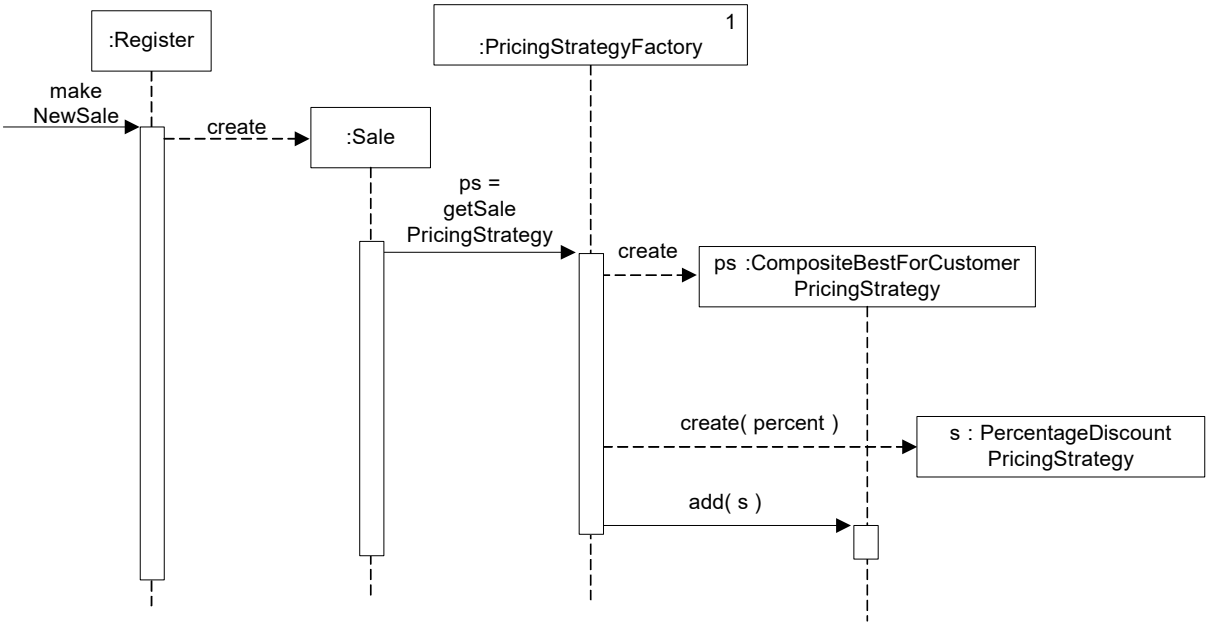


When do we create SalePricingStratigies?

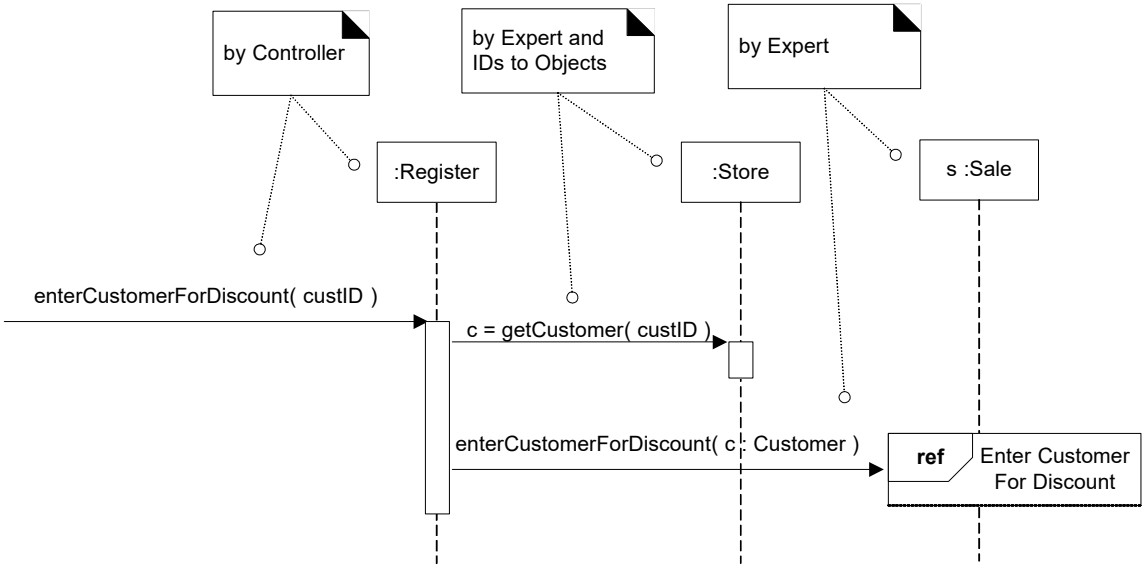
There are three points in the scenario where pricing strategies may be added to the composite:

1. Current store-defined discount, added when the sale is created
2. Customer type discount, added when the customer type is communicated to the POS
3. Product type discount (if bought Darjeeling tea, 15% off the overall sale), added when the product is entered to the sale

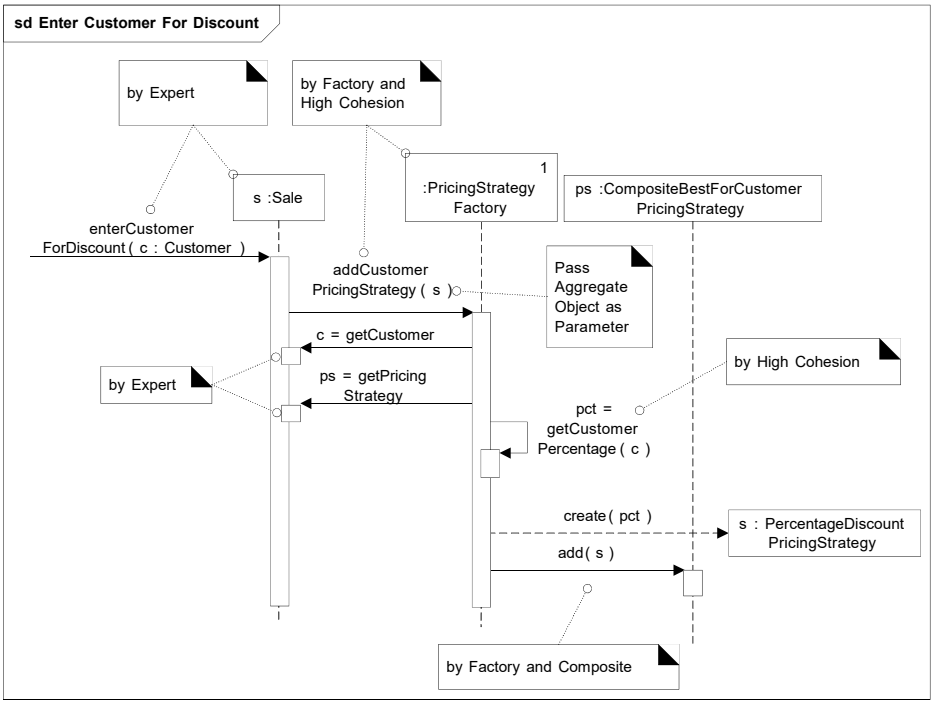
1. Current store-defined discount, added when the sale is created



2. Customer type discount, added when the customer type is communicated to the POS



2. Customer type discount, added when the customer type is communicated to the POS (Cont'd)



Façade Pattern

Problem

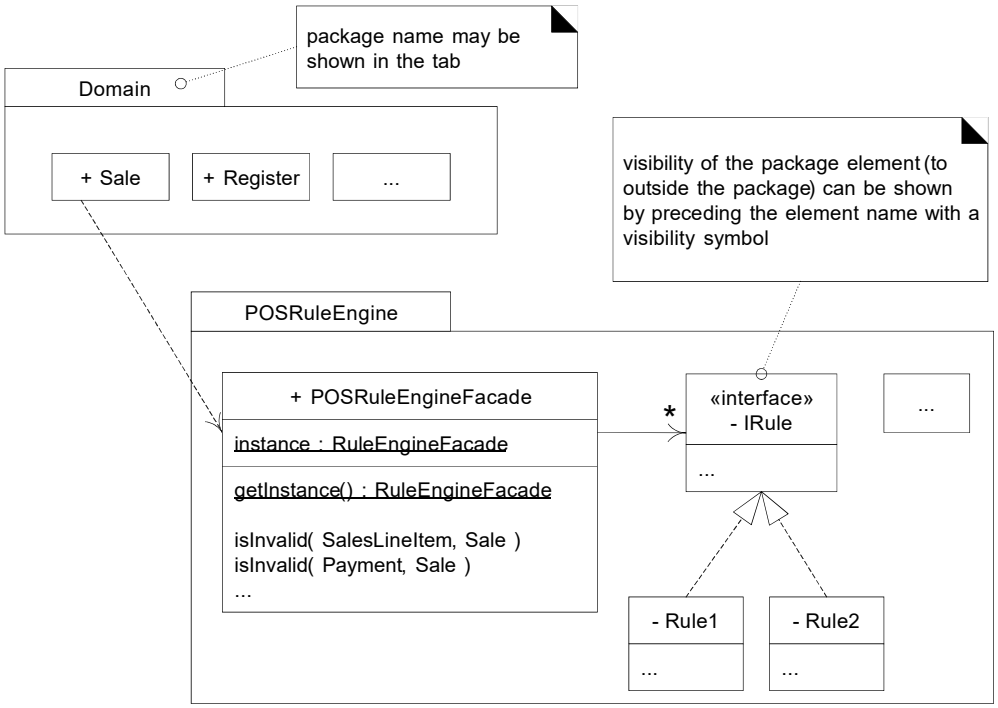
A common, unified interface to a disparate set of implementations or interfaces such as within a subsystem is required. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change. What to do?

Solution

Define a single point of contact to the subsystem, a facade object that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components

We will define a “pluggable rule engine" subsystem, whose specific implementation is not yet known. It will be responsible for evaluating a set of rules against an operation. The architect also wants to design for a separation of concerns, and factor out this rule handling into a separate subsystem

Façade Pattern (Cont’d)



Observer/Publish-Subscribe Pattern

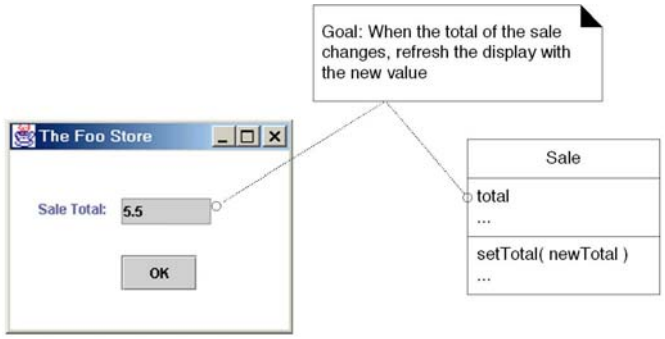
Problem

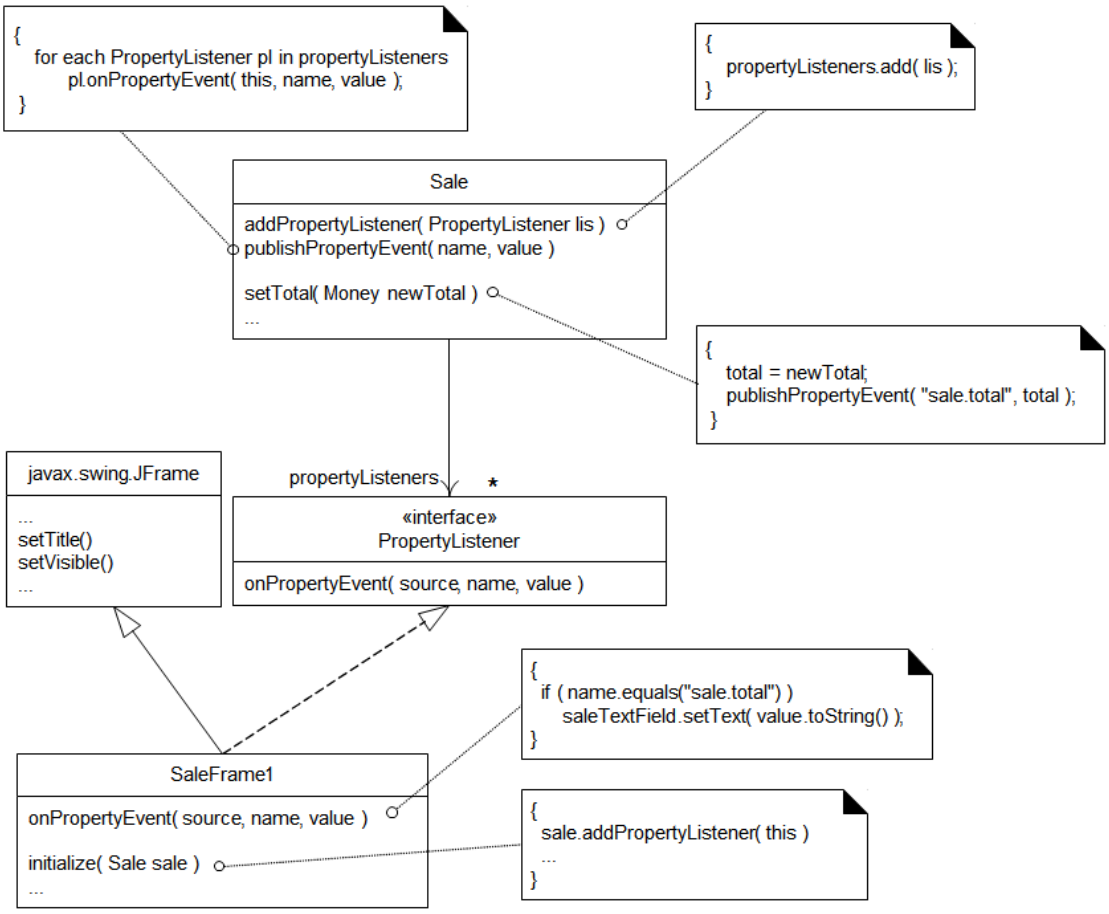
Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?

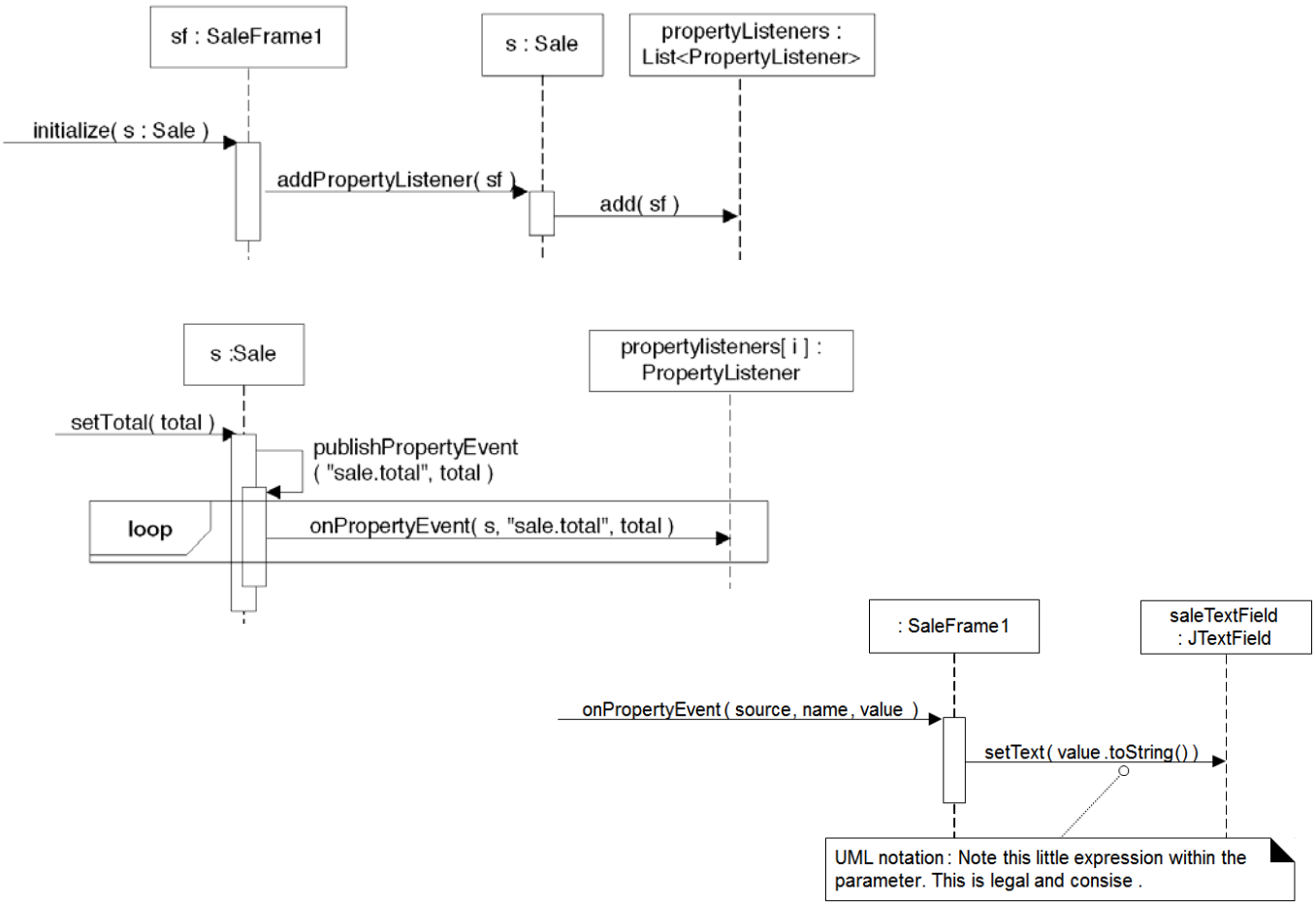
Solution

Define a "subscriber" or "listener" interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

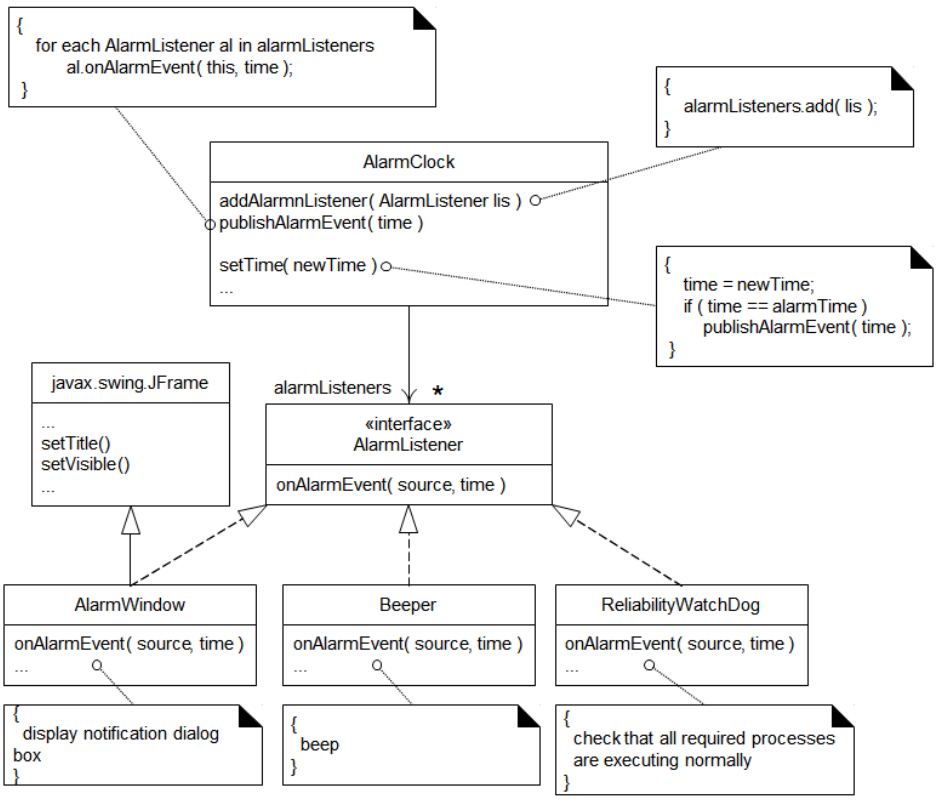
Another requirement for the iteration is adding the ability for a GUI window to refresh its display of the sale total when the total changes



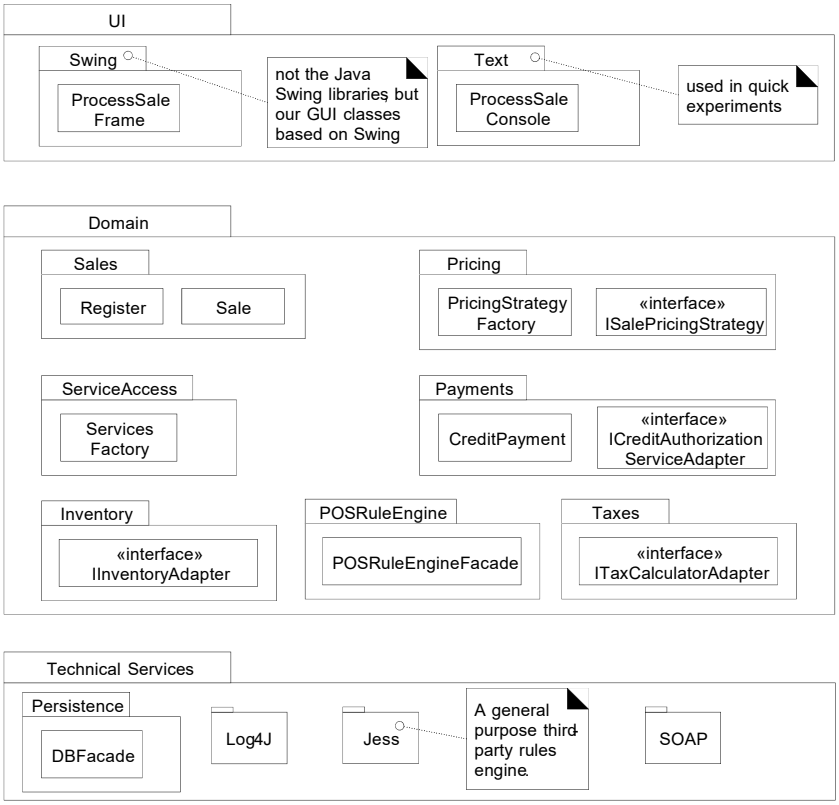




Other Usage of Observer Pattern



NextGen POS System Architecture



Advanced Principles of Object-Oriented Design

SRP: The Single Responsibility Principle

OCP: The Open Closed Principle

LSP: The Liskov Substitution Principle

ISP: The Interface Segregation Principle

DIP: The Dependency Inversion Principle